intel®

APPLICATION NOTE

AP-235

1

October 1990

An 82586 Data Link Driver

CHARLES YAGER

Order Number: 231421-002

AN 82586 DATA LINK DRIVER

CONTENTS

PAGE

INTRODUCTION

This application note describes a design example of an IEEE 802.2/802.3 compatible Data Link Driver using the 82586 LAN Coprocessor. The design example is based on the "Design Model" illustrated in "Programming the 82586". It is recommended that before reading this application note, the reader clearly understands the 82586 data structures and the Design Model given in "Programming the 82586".

"Programming the 82586" discusses two basic issues in the design of the 82586 data link driver. The first is how the 82586 handler fits into the operating system. One approach is that the 82586 handler is treated as a "special kind of interface" rather than a standard I/O interface. The special interface means a special driver that has the advantage of utilizing the 82586 features to enhance performance. However the performance enhancement is at the expense of device dependent upper layer software which precludes the use of a standard I/O interface.

The second issue "Programming the 82586" discusses is which algorithms to choose for the CPU to control the 82586. The algorithms used in this data link design are taken directly from "Programming the 82586". Command processing uses a linear static list, while receive processing uses a linear dynamic list.

The application example is written in C and uses the Intel C compiler. The target hardware for the Data Link Driver is the iSBC 186/51 COMMputer, however a version of the software is also available to run on the LANHIB Demo board.

1.0 FITTING THE SOFTWARE INTO THE OSI MODEL

The application example consists of four software modules:

- Data Link Driver (DLD): drives the 82586, also known as the 82586 Handler.
- Logical Link Control (LLC): implements the IEEE 802.2 standard.
- User Application (UAP): exercises the other software modules and runs a specific application.
- C hardware support: written in assembly language, supports the Intel C compiler for I/O, interrupts, and run time initialization for target hardware.

Figure 1 illustrates how these software modules combined with the 82586, 82501 and 82502 complete the first two layers of the OSI model. The 82502 implements an IEEE 802.3 compatible transceiver, while the 82501 completes the Physical layer by performing the serial interface encode/decode function.

The Data Link Layer, as defined in the IEEE 802 standard documents, is divided into two sublayers: the Logical Link Control (LLC) and the Medium Access Control (MAC) sublayers. The Medium Access Control sublayer is further divided into the 82586 Coprocessor plus the 82586 Handler. On top of the MAC is the LLC software module which provides IEEE 802.2 compatibility. The LLC software module implements the Station Component responses, dynamic addition and deletion of Service Access Points (SAPs), and a class 1 level of service. (For more information on the LLC sublayer, refer to IEEE 802.2 Logical Link Control Draft Standard.) The class 1 level of service provides a connectionless datagram interface as opposed to the class 2 level of service which provides a connection oriented level of service similar to HDLC Asynchronous Balanced Mode.

On top of the Data Link Layer is the Upper Layer Communications Software (ULCS). This contains the Network, Transport, Session, and Presentation Layers. These layers are not included in the design example, therefore the application layer of this ap note interfaces directly to the Data Link layer.

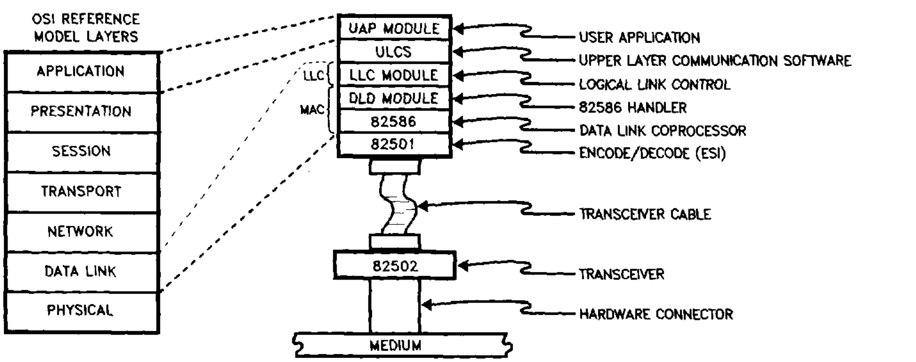

Figure 1. Data Link Driver's Relationship to OSI Reference Mode 1

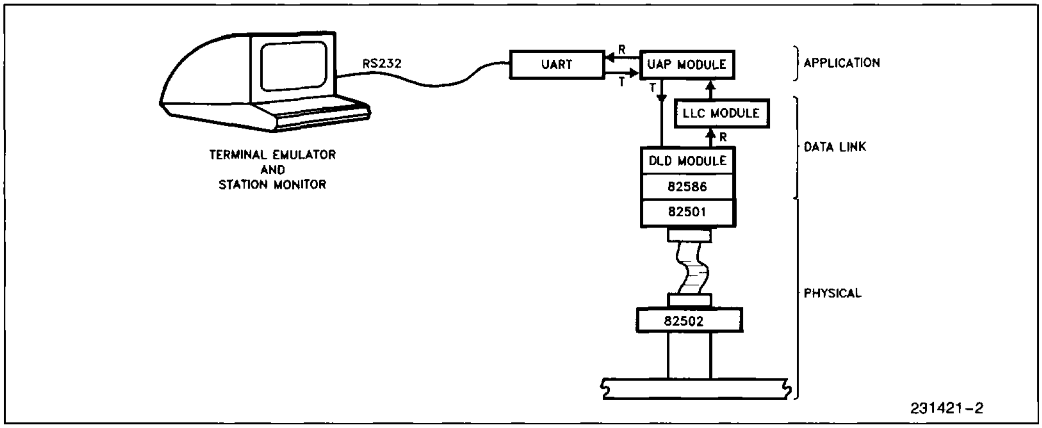

Figure 2. Block Diagram of the Hardware and Software

The application layer is implemented in the User Application (UAP) software module. The UAP module operates in one of three modes: Terminal Mode, Monitor Mode, and High Speed Transmit Mode. The software initially enters a menu driven interface which allows the program to modify several network parameters or enter one of the three modes.

The Terminal Mode implements a virtual terminal with datagram capability (connectionless "class 1" service). This mode can also be thought of as an async to IEEE 802.3/802.2 protocol converter.

The Monitor Mode provides a dynamic update on the terminal of 6 station related parameters. While in the monitor mode, any size frame can be repeatedly transmitted to the cable in a software loop.

High Speed Transmit Mode transmits frames to the cable as fast as the software possibly can. This mode demonstrates the throughput performance of the Data Link Driver.

The UAP gathers network statistics in all three modes as well as when it is in the menu. In addition, the UAP module provides the capability to alter MAC and LLC addresses and re-initialize the data link. (Figure 2 shows a combined software and hardware block diagram.)

2.0 LARGE MODEL COMPILATION

All the modules in this design example are compiled under the Large Model option. This has the advantages of using the entire 1 Mbyte address space, and allowing the string constants to be stored in ROM. In the Large Model it is important to consider that the 82586's data structures, SCB, CB, TBD, FD, and RBD, must reside within the same data segment. This data segment is determined at locate time.

The C__Assy__Support module has a run time start off function which loads the DLD data segment into a global variable SEGMT__. This data segment is used by the 82586 Handler for address translation purposes. The 82586 uses a flat address while the 80186 uses a segmented address. Any time a conversion between 82586 and 80186 addresses are needed the SEGMT__ variable is used.

Pointers for the 80186 in the large model are 32 bits, segment and offset. All the 82586 link pointers are 16 bit offsets. Therefore when trading pointers between the 82586 and the 80186, two functions are called: Offset (ptr), and Build__Ptr (offset). Offset (ptr) takes a 32 bit 80186 pointer and returns just the offset portion for the 82586 link pointer. While Build__Ptr (offset) takes an 82586 link pointer and returns a 32 bit 80186 pointer, with the segment part being the SEGMT__ variable. Offset ( ) and Build__Ptr( ) are simple functions written in assembly language included in the C__ Assy__Support module.

In the small model, Offset ( ) and Build__Ptr( ) are not needed, but the variable SEGMT__ is still needed for determining the SCB pointer in the ISCP, and in the Transmit and Receive Buffer Descriptors.

3.0 THE 82586 HANDLER

3.1 The Buffer Model

The buffer model chosen for the 82586 Handler is the "Design Model" as described in "Programming the 82586". This is based on the 82586 driver as a special driver rather than as a standard driver. Using this approach the ULCS directly accesses the 82586's Transmit and Receive Buffers, Buffer Descriptors and Frame Descriptors. This eliminates buffer copying. Transmit and receiver buffer passing is done entirely through pointers.

The only hardware dependencies between the Data Link and ULCS interface are the buffer structures. The ULCS does not handle the 82586's CBs, SCB or initialization structures. To isolate the data link interface from any hardware dependencies while still using the design model, another level of buffer copying must be introduced. For example, when the ULCS transmits a frame it would have to pass its own buffers to the data link. The data link then copies the data from ULCS buffers into 82586 buffers. When a frame is received, the data link copies the data from the 82586's buffers into the ULCS buffers. The more copying that is done the slower the throughput. However, this may be the only way to fit the data link into the operating system. The 82586 Handler can be made hardware independent by adding a receive and transmit function to perform the buffer copying.

The 82586 Handler allocates buffers from two pools of memory: the Transmit pool, and the Receive pool as illustrated in Figure 3. The Transmit pool contains Transmit Buffer Descriptors (TBDs) and Transmit Buffers (TBs). The Receive pool contains Frame Descriptors (FDs), Receive Buffer Descriptors (RBDs), and Receive Buffers (RBs).

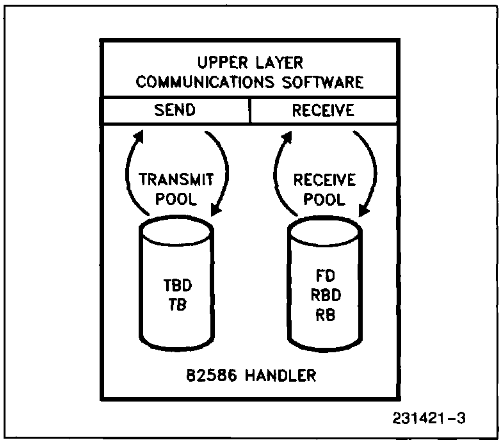

Figure 3. 82586 Handler Memory Management Model

When the ULCS wants to transmit, it requests a TBD from the handler. The handler returns a pointer to a free TBD. Each TBD has a TB attached to it. The ULCS fills the buffer, sets the appropriate fields in the TBD, and passes the TBD pointer back to the handler for transmission. After the frame is transmitted, the handler places the TBD back into the free TBD pool. If the ULCS needs more than one buffer per frame, it simply requests another TBD from the handler and performs the necessary linkage to the previous TBD.

On the receive side, the RFA pool is managed by the 82586 itself. When a frame is received, the 82586 interrupts the handler. The handler passes a FD pointer to the ULCS. Linked to the FD is one or more RBDs and RBs. The ULCS extracts what it needs from the FD, RBDs and RBs, and returns the FD pointer back to the handler. The handler places the FD and RBDs back into the free RFA pool.

3.2 The Handler Interface

The handler interface provides the following basic functions:

- initialization
- sending and receiving frames
- adding and deleting multicast addresses
- getting transmit buffers
- returning receive buffers

Figure 4 lists the Handler Interface functions.

On power up, the initialization function is called. This function initializes the 82586, and performs diagnostics. After initialization, the handler is ready to transmit and receive frames, and add and delete multicast addresses.

To send a frame, the ULCS gets one or more transmit buffers from the handler, fills them with data, and calls the send function. When a frame is received, the handler calls a receive function in the ULCS. The ULCS receive function removes the information it needs and returns the receive buffers to the handler. The addition and deletion of multicast addresses can be done "on the fly" any time after initialization. The **receiver doesn't have to be disabled** when this is done.

The command interface to the handler is totally asynchronous—the ULCS can issue transmit commands or multicast address commands whenever it wants. The commands are queued by the handler for the 82586 to execute. If the command queue is full, the send frame procedure returns a false status rather than true. The size of the command queue can be set at compile time by setting the CB—CNT constant. Typically the command queue never has more than a few commands on it because the 82586 can execute commands faster than the ULCS can issue them. This is not the case in a heavily loaded network when deferrals, collisions, and retries occur.

The command interface to the 82586 handler is hardware independent; the only hardware dependence is the buffering. A hardware independent command interface doesn't have any performance penalty, but some 82586 programmability is lost. This shouldn't be of concern since most data links do not change configuration parameters during operation. One can simply modify a few constants and recompile to change frame and network parameters to support other data links.

1

Handler Interface Functions	Description
Init_586( )	Initialize the Handler
Send_Frame (ptbd, padd)	Sends a frame to the cable. ptbd—Transmit Buffer Descriptor pointer padd—Destination Address pointer
Recv_Frame (pfd)	Handler calls this function which resides in the ULCS. pfd_Frame Descriptor pointer
Add_Multicast_Address (pma)	Adds one multicast address pma—Multicast Address pointer
Delete_Multicast_Address (pma)	Deletes one multicast address
Get_Tbd( )	Get a Transmit Buffer Descriptor pointer
Put_Free_Rfa (pfd)	Returns a Frame Descriptor and Receive Buffer Descriptors to the 82586.

Figure 4. List of Handler Interface Functions

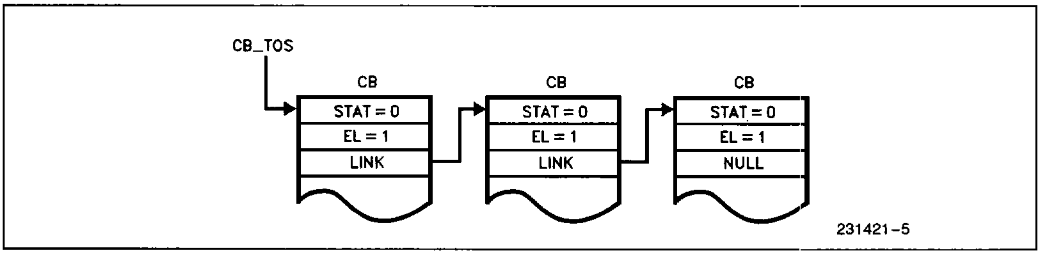

Figure 5. Free CB Pool

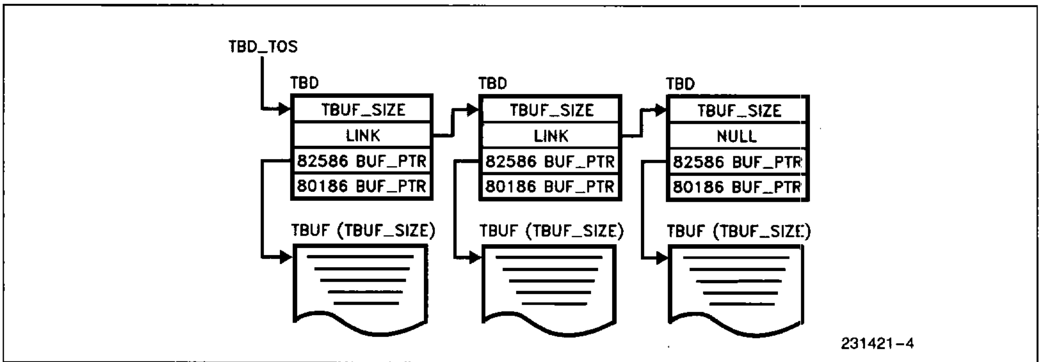

Figure 6. Free Transmit Buffer Descriptor Pool

3.3 Initialization

The function which initializes the 82586 handler, Init__586( ), is called by the ULCS on power up or reinitialization. **Before this function is called, an 82586 hardware or software reset should occur.** The Initialization occurs in three phases. The first phase is to initialize the memory. This includes flags, vectors, counters, and data structures. The second phase is to initialize the 82586. The third phase is to perform self test diagnostics. Init__586( ) returns a status byte indicating the results of the diagnostics.

Init__586( ) begins by toggling the 82501 loopback pin. If the 82501 is powered up in loopback, the $\overline{CRS}$ and $\overline{CDT}$ pin may be active. To reset this condition, the loopback pin is toggled. The 82501 should remain in loopback for the first part of the initialization function.

Phase 1 executes initialization of all the handlers flags, interrupt vectors, counters, and 82586 data structures. There are two separate functions which initialize the CB and RFA pools: Build__CB( ) and Build__Rfa( ).

3.3.1 BUILDING THE CB AND RFA POOLS

Build__CB( ) builds a stack of free linked Command Blocks, and another stack of free linked Transmit Buffer Descriptors. (See Figures 5 and 6.) Each stack has a Top of Stack pointer, which points to the next free structure. The last structure on the list has a NULL link pointer.

The CBs within the list are initialized with 0 status, EL bit set, and a link to the next CB. The TBD structures are initialized with the buffer size, which is set at compile time with the TBUF__SIZE constant, a link to the next TBD, and an 82586 pointer to the transmit buffer. This pointer is a 24 bit flat/physical address. The address is built by taking the transmit buffer's data segment address, shifting it to the left by 4 and adding it to the transmit buffer offset. An 80186 pointer to the transmit buffer is added to the TBD structure so that the 80186 does not have to translate the address each time it accesses the transmit buffer.

Build__Rfa( ) builds a linear linked Frame Descriptor list and a Receive Buffer Descriptor list as shown in Figure 7. The status and EL bits for all the free FDs are 0. The last FD's EL bit is 1 and link pointer is NULL. The first FD on the FD list points to the first RBD on the RBD list. The RBDs are initialized with both 82586 and 80186 buffer pointers. The 80186 buffer pointer is added to the end of the RBD structure. Begin and end pointers are used to mark the boundaries of the free lists.

3.3.2 82586 INITIALIZATION

The 82586 initialization data structure SCP is already set since it resides in ROM, however, the ISCP must be loaded with information. Within the SCP ROM is the pointer to the ISCP; the ISCP is the only absolute address needed in the software. Once the ISCP address is determined, the ISCP can be loaded. The SCB base is obtained from the C__Assy__Support module. The global variable SEGMT__contains the address of the

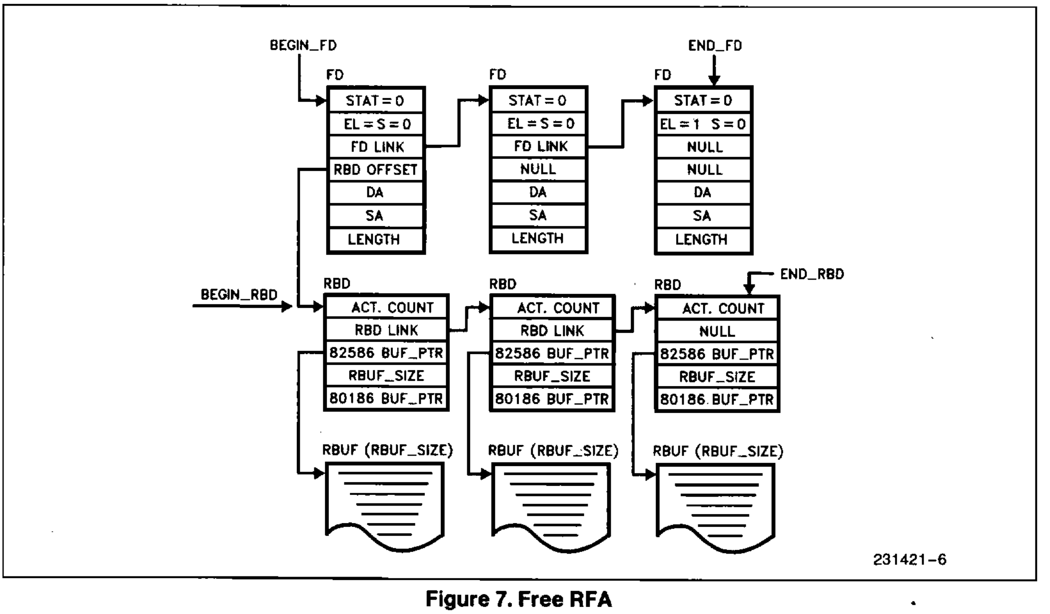

Figure 7. Free RFA

data segment of the handler. The 80186 shifts this value to the left by 4 and loads it into the SCB base. The SCB offset is now determined by taking the 32 bit SCB pointer and passing it to the Offset( ) function.

The 82586 interrupt is disabled during initialization because the interrupt function is not designed to handle 82586 reset interrupts. To determime when the 82586 is finished with its reset/initialization, the SCB status is polled for both the CX and CNA bits to be set. After the 82586 is initialized, both the CX and CNA interrupts are acknowledged.

The 82586 is now ready to execute commands. The Configuration is executed first to place the 82586 in internal loopback mode, followed by the IA command. The address for the IA command is read off of a prom on the PC board.

3.3.3 SELF TEST DIAGNOSTICS

The final phase of the handler initialization is to run the self test diagnostics. Four tests are executed: Diagnose command, Internal loopback, External loopback through the 82501, and External loopback through the transceiver. If these four tests pass, the data link is ready to go on line.

The function that executes these diagnostics is called Test__Link( ). If any of the tests fail, Test__Link( ) returns immediately with the Self__Test global variable set to the type of failure. This Self__Test global variable is then returned to the function which originally called Init__586( ). Therefore Init__586( ) can return one of five results: FAILED__DIAGNOSE, FAILED__LPBK__INTERNAL, FAILED__LPBK__EXTERNAL, FAILED__LPBK__TRANSCEIVER or PASSED.

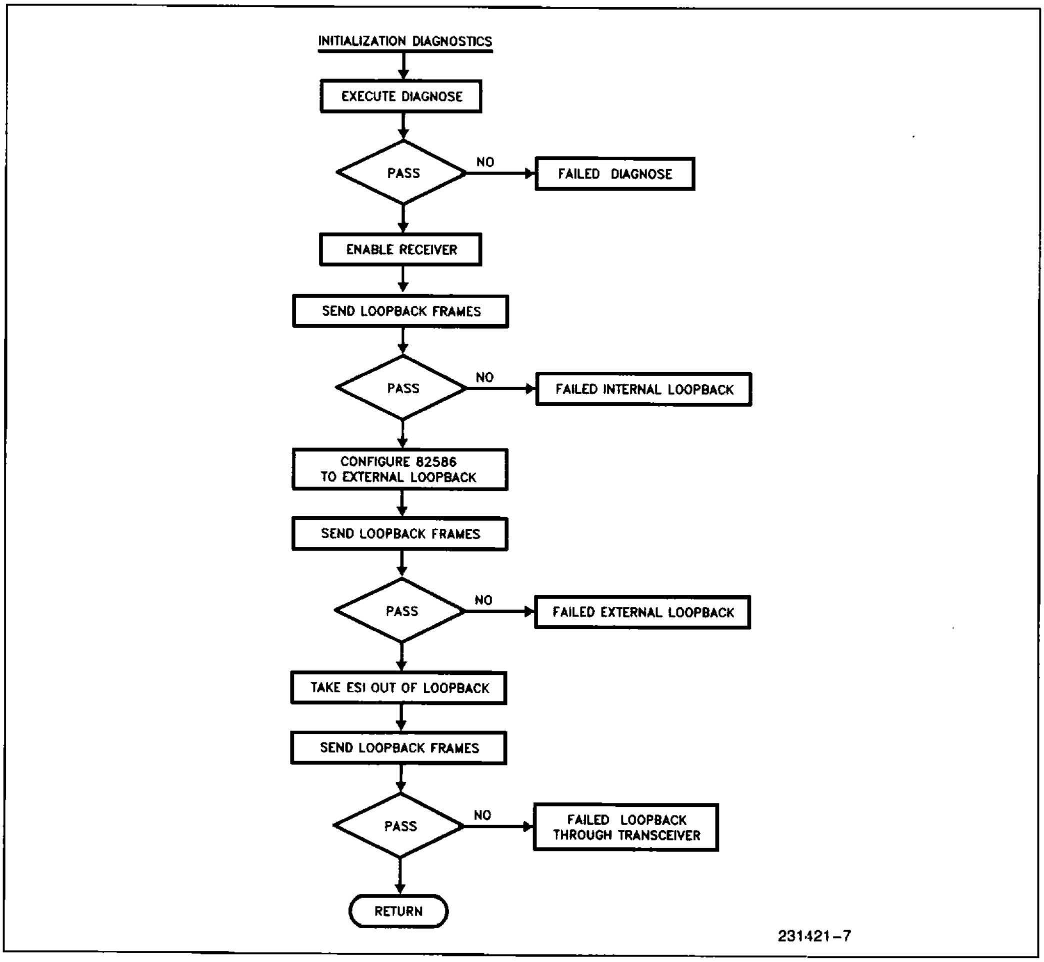

Figure 8. Initialization Diagnostics: Test__Link ()

The Diagnose( ) function, called by Test_Link( ), does not return until the diagnose command is completed. If the interrupt service routine detects that a Diagnose command was completed then it sets a flag to allow the Diagnose( ) function to return, and it also sets the Self_Test variable to FAIL if the Diagnose command failed. If the Diagnose command completed successfully, the loopback tests are performed.

Before any loopback tests are executed, the Receive Unit is enabled by calling Ru_Start( ). Loopback tests begin by calling Send_Lpbk_Frame( ), which sends 8 frames with known loopback data and its own destination address. More than one loopback frame is sent in case one or more of them are lost. Also several of the frames will have been received by the time flags.lpbk_test is checked.

Two flag bits are used for the loopback tests: flags.lpbk_mode, and flags.lpbk_test. flags.lpbk_mode is used to indicate to the receive section that the frames received are potentially loopback frames. The receive section will pass receive frames to the Loopback Check( ) function if the flags.lpbk_mode bit is set. The Loopback_Check( ) function first compares the source address of the frame with its station address. If this matches then the data is checked with the known loopback data. If the data matches, then the flags.lpbk_test bit is set, indicating a successful loopback. The flow of the Test_Link( ) function is displayed in Figure 8.

3.4 Command Processing

Command blocks are queued up on a static list for the 82586 to execute. The flow of a command block is given in Figure 9. When the handler executes a command it first has to get a free command block. It does this by calling Get_CB( ) which returns a pointer to a free command block. The CB structure is a generic one in which all commands except the MC-Setup can fit in. The handler then loads into the CB structure the type of command and associated parameters. To issue the command to the 82586 the Issue_CU_Cmd( ) function is called with the pointer to the CB passed to this function. Issue_CU_Cmd( ) places the command on

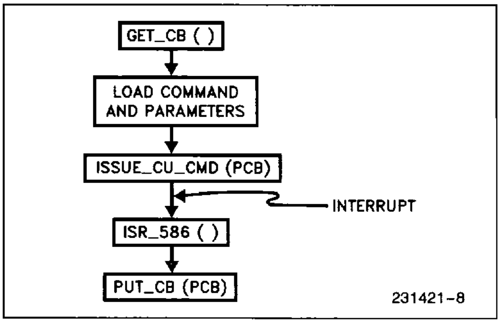

Figure 9. The Flow of a Command Block

the 82586's static command block list. After the 82586 executes the command, it generates an interrupt. The interrupt routine, Isr_586( ), processes the command and returns the Command Block to the free command block list by calling Put_Cb( ).

3.4.1 ACCESSING COMMAND BLOCKS-GET_CB() and PUT_CB()

Get_Cb( ) returns a pointer to a free command block. The free command blocks are in a linear linked list structure which is treated as a stack. The pointer cb_tos points to the next available CB. Each time a CB is requested, Get_Cb( ) pops a CB off the stack. It does this by returning the pointer of cb_tos. cb_tos is then updated with the CB's link pointer. When the CB list is empty, Get_Cb( ) returns NULL.

There are two types of nulls, the 82586 'NULL' is a 16 bit offset, OFFFFH, in the 82586 data structures. The 80186 null pointer, 'pNULL', is a 32 bit pointer; with OFFFFH offset and the 82586 handler's data segment, SEGMT_, as the base.

Put_Cb( ) pushes a free command block back on the list. It does this by placing the cb_tos variable in the returned CB's link pointer field, then updates cb_tos with the pointer to the returned CB.

3.4.2 ISSUING CU COMMANDS-ISSUE_CU_CMD()

This function queues up a command for the 82586 to execute. Since static lists are used, each command has its EL bit set. There is a begin_cbl pointer and an end_cbl pointer to delineate the 82586's static list. If there are no CBs on the list, then begin_cbl is set to pNULL. (Figure 10 illustrates the static list.) Each time a command is issued, a deadman timer is set. When the 82586 interrupts the CPU with a command completed, the deadman timer is reset.

Issue_Cu_Cmd( ) begins by disabling the 82586's interrupt. It then determines whether the list is empty or not. If the list is empty, begin and end pointers are loaded with the CB's address. The CU must then be started. Before a CU_START can be issued, the SCB's cbl_offset field must be loaded with the address of the command, the Wait_Scb( ) function must be called to insure that the SCB is ready to accept a command, and the deadman timer must be initialized. If the list is not empty, then the command block is queued at the end of the list, and the interrupt service routine Isr_586( ), will continue generating CAs for each command linked on the CB list until the list is empty.

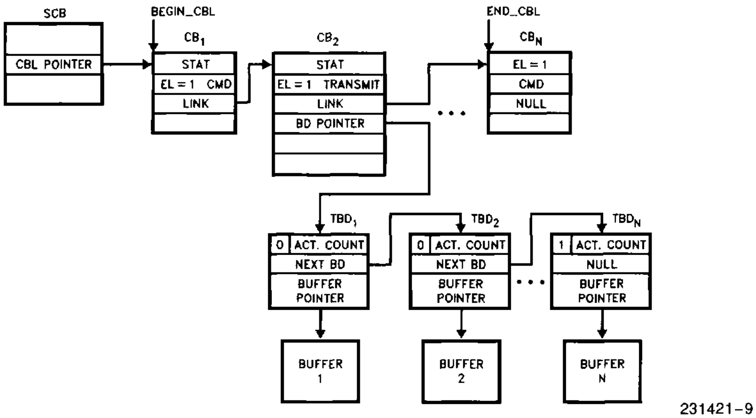

Figure 10. The Static Command Block List

3.4.3 INTERRUPT SERVICE ROUTINE-ISR__586()

Isr__586( ) starts off by saving the interrupts that were generated by the 82586 and acknowledging them. Acknowledgment must be done immediately because if a second interrupt were generated before the acknowledgment, the second interrupt would be missed. The interrupt status is then checked for a receive interrupt and if one occurred the Recv__Int__Processing( ) function is called. After receive processing is check the CPU checks whether a command interrupt occurred. If one did, then the deadman timer is reset and the results of the command are checked. There are only two particular commands which the interrupt results are checked for: Transmit and Diagnose. The Diagnose command needs to be tested to see if it passed, plus the diagnose status flag needs to be set so that the initialization process can continue.

The transmit command status provides network management and station diagnostic information which is useful for the "Network Management" function of the ISO model. The following statistics are gathered in the interrupt routine: good__transmit__cnt, sqe__err__cnt, defer__cnt, no__crs__cnt, underrun__cnt, max__col__cnt. To speed up transmit interrupt processing a flag is tested to determine whether these statistics are desired, if not this section of code is skipped.

The sqe error requires special considerations when used for statistic gathering or diagnostics. The sqe status bit indicates whether the transceiver passed its self test or not. The transceiver executes a self test after each transmission. If the transceiver's self test passed, it will activate the collision signal during the IFS time.

The sqe status bit will be set if the transceiver's self test passed. However if the sqe status bit is not set, the transceiver may still have passed its self test. Several events can prevent the sqe bit from being set. For example, the first transmit command status after power up will not have the sqe bit set because the sqe is always from the previous command. Also if any collisions occur, the sqe bit might not be set. This has to do with the timing of when the sqe signal comes from the transceiver. It is possible that a JAM signal from a remote station can overlap the sqe signal in which case the 82586 will not set the sqe status bit. Therefore the sqe error count should only be recorded when no collisions occur.

One other situation can occur which will prevent the SQE status bit from being set. If transmit command reaches the maximum retry count, the next transmit command's SQE bit will not be set.

The final phase of interrupt command processing determines if another command is linked, and returns the CB to the free command block list. Another command being linked is indicated by the CB link field not being NULL. In this case the deadman timer and the 82586's CU are re-started. If the CB link is NULL, there are no further commands to execute, and begin__cbl is set to pNULL.

3.4.4 SENDING FRAMES-SEND__FRAME (PTBD, PADD)

Send__Frame( ) receives two parameters, a pointer to the first Transmit Buffer Descriptor, and a pointer to the destination address. There may be one or more TBDs attached. The last TBD is indicated by its link

field being NULL and the EOF bit set. It is the responsibility of the ULCS to make sure this is done before calling Send__Frame( ).

Send__Frame( ) begins by trying to obtain a command block. If the free command block list is empty, the send frame function returns with a false result. It is up to the ULCS to either continue attempting transmission or attempt at a later time. The send frame function calculates the length field by summing up the TBDs actual count field. After the length field is determined, send frame checks to see if padding is required. If padding is necessary, Send Frame will change the act count field in the TBD to meet the minimum frame requirements. This technique transmits what ever was in the buffer as padding data. If security is an issue, the padding data in the buffer should be changed.

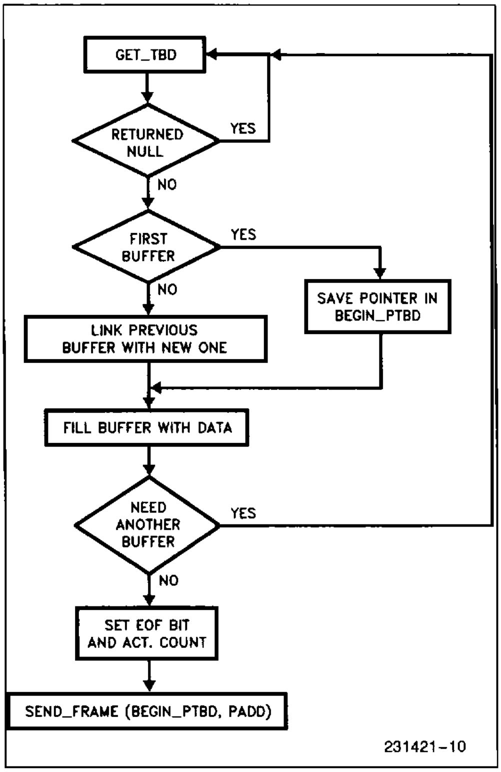

Figure 11. Flow Chart for Sending a Frame

3.4.5 ACCESSING TRANSMIT BUFFERS-GET__TBD() AND PUT__TBD()

Get__Tbd( ) returns a pointer to a free Transmit Buffer Descriptor, and Put__Tbd( ) returns one or more linked Transmit Buffer Descriptors to the free list. The TBD which Get__Tbd( ) allocates has its link pointer set to NULL, and its EOF bit cleared. If another buffer is needed, the link field in the old TBD must be set to point to the new TBD. The last TBD used should have its link pointer set to NULL and its EOF bit set. Figure 11 shows the flow chart of getting buffers and sending a frame.

Put__Tbd (ptbd) is called by the Isr__586( ) function when the 82586 is done transmitting the buffers. A pointer to the first TBD is passed to Put__Tbd( ). Put__Tbd( ) finds the end of the list of TBDs and returns them to the free buffer list.

3.4.6 MULTICAST ADDRESSES

The 82586 handler maintains a table of multicast addresses. Initially this table is empty. To enable a multicast address the Add__Multicast__Address(pma) function is called; to disable a multicast address, Delete__Multicast__Address(pma) function is called. Both functions accept a parameter which points to the multicast address. Add and Delete functions perform linear searches through the Multicast Address Table (MAT).

Add scans the entire MAT once to check if the address being added is a duplicate of one already loaded. Add will not enter a duplicate muilticast address. If there are no duplicates Add goes to the beginning of the MAT and looks for a free location. If it finds one, it loads the new address into the free location and sets the location status to INUSE. If no free locations are available, Add returns a false result.

Delete looks for a used location in the MAT. When it finds one, it compares the address in the table with the address passed to it. If they match, the location status is set to FREE and a TRUE result is returned. If no match occurs, the result returned is FALSE.

If Add or Delete change the MAT, they update the 82586 by calling Set__Multicast__Address( ). This function executes an 82586 MC Setup command. Set__Mulitcast__Address( ) uses the addresses in the MAT to build the MC Setup command. The MC Setup command is too big to be built from the free CBs. Free CB

command blocks are 18 bytes long, while the MC Setup command can be up to 16,392 bytes. Therefore a separate Multicast Address Command Block (ma_cb) must be allocated and used. The size of the ma_cb and MAT are determined at compile time based on the MULTI_ADDR_CNT constant. The design example allows up to 16 multicast addresses.

Since there is only one ma_cb, and it is not compatible with the other CBs, it must be treated differently. Only one ma_cb can be on the 82586 command list. The ma_cb command word is used as a semaphore. If it is zero, the command is available. If not, Set_Multicast_Address( ) must wait until the ma_cb is free. Also the interrupt routine can't return the ma_cb to the free CB list. It just clears the cmd field, to indicate that ma_cb is available.

The 82586's receiver does not have to be disabled to execute the MC Setup command. If the 82586 is receiving while this command is accessed, the 82586 will finish reception before executing the MC Setup comand. If the MC Setup command is executing, the 82586 automatically ignores incoming frames until the MC Setup is completed. Therefore multicast addresses can be added and deleted on the fly.

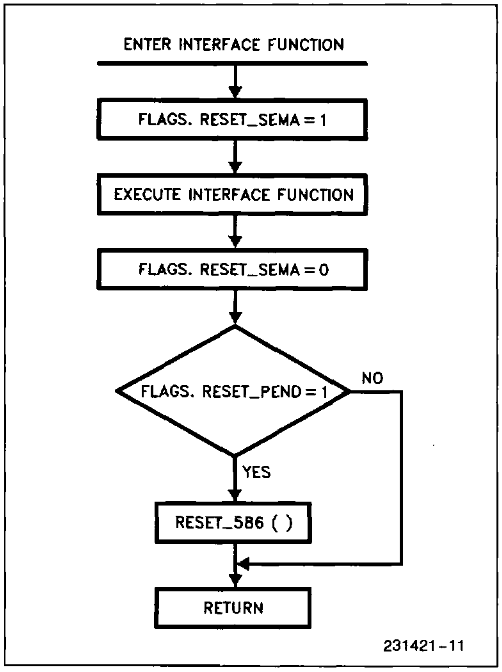

Figure 12. Reset Semaphore

3.4.7 RESETTING THE 82586-RESET_586()

The 82586 rarely if ever locks up in a well behaved network; (i.e. one that obeys IEEE 802.3 specifications). The lock-ups identified were artificially created and would normally not occur. This data link driver has been tested in an 8 station network under various loading conditions. No lock-ups occurred under any of the data link drivers test conditions. However the reset software has been tested by simulating a lockup. This can be done by having the 82586 transmit, and disabling the CTS pin for a time longer than the deadman timer.

An 82586 deadlock is not a fatal error. The handler is designed to recover from this problem. As mentioned before, each time the 82586 is given a CA to begin executing a command, a deadman timer is set. The deadman timer is reset when a CNR interrupt is generated. If the CNR interrupt is not generated before the deadman timer expires, the 82586 must be reset.

Resetting of the 82586 should not be done while the handler software is executing. This could create a software deadlock by interrupting a critical section of code in the handler. To insure that the Reset_586( ) function is not executed while the handler is executing, all of the entry points to the handler (i.e. interface functions) set a semaphore flag bit called flags.reset_sema. This flag is cleared when the interface functions are exited.

If the Deadman timer interrupt occurs while flags.reset_sema is set, another flag is set (flag.reset_pend) indicating that the Reset_586( ) function should be called when the interface functions are exited. However if the deadman timer interrupt occurs when flags.reset_sema is clear, Reset_586( ) is called immediately. Figure 12 shows the logic for entering and exiting interface functions.

Reset_586( ) begins by disabling the 82586 interrupt, placing the ESI in loopback, and resetting the 82586. The reset can be a software or a hardware reset. However, there are certain lockups in the 82586 where only a hardware reset will suffice. (The 82586 errata sheet explicitly indicates which deadlocks require a hardware reset.) After the reset, Reset_586( ) executes a Configure, IA-Setup, and a MC-Setup command; the MC_Setup command is built from the multicast address table (MAT). The 82586 Command Queues and Receive Frame Queues are left untouched so that the 82586 can continue executing where it left off before the deadlock. This way no frames or commands are lost. This requires that a separate reset CB and reset Multicast CB is used, because other CBs already in use cannot be disturbed.

3.5 Receive Frame Processing

The following functions are used for Receive Frame Processing:

Recv_Int_Processing( )	Called by Isr_586( ) to remove FDs and RBDs from the 82586's RFA
Recv_Frame (pfd)	Called by Recv_Int_Processing( ). This function resides in the ULCS
Check_Multicast (pfd)	Used for perfect Multicast filtering
Put_Free_Rfa (pfd)	Returns FDs and RBDs to the 82586's RFA
Ru_Start( )	Restarts the RU when in the IDLE or No Resources state.

3.5.1 RECEIVE INTERRUPT PROCESSING-RECV_INT_PROCESSING()

The Recv_Int_Processing( ) function is called by Isr_586( ) when the FR bit in the SCB is set. The Recv_Int_Processing( ) function checks whether any FDs and RBDs on the free list have been used by the 82586. If they have, Recv_Int_Processing( ) removes the used FDs and RBDs from the free list, and passes them to the ULCS.

The Recv_Int_Processing( ) function is a loop where each pass removes a frame from the 82586's RFA. When there are no more used FDs and RBDs on the RFA, the function calls RU_Start( ), then returns to Isr_586( ). The first part of the loop checks to see if the C bit in the first FD of the free FD list is set. If the C bit is set, the function determines if one or more RBDs are attached. If there are RBDs attached, the end of the RBD list is found. The last RBD's link field is used to update begin_rbd pointer, and then it's set to NULL.

After the receive frame has been delineated from the RFA, some information about the frame is needed to determine which function to pass it to. Since the save bad frame configure bit is not set, the only bad frame on the list could be an out of resource frame. An out of resource frame is returned to the RFA by calling Put_Free_RFA (pfd). If the flags.lpbk_mode bit is set, the frame is given to the loopback check function. If the destination address of the frame indicates a multicast, the check multicast function is called. If the frame has passed all of the above tests and still has not been returned, it is passed to the Recv_Frame( ) function which resides in the ULCS.

Check_Multicast (pfd) determines whether the multicast address received is in the multicast address table. This is necessary because the 82586 does not have perfect multicast address filtering. Check_Multicast does a byte by byte comparison of the destination address with the addresses in the multicast address table. If no match occurs, it returns false, and Recv_Int_Processing calls Put_Free_RFA( ) to return the frame to the RFA. If there is a match, Check_Multicast( ) returns TRUE and Recv_Int_Processing( ) calls Recv_Frame( ), passing the pointer to the FD of the frame received.

3.5.2 RETURNING FDs AND RBDs-PUT_FREE_RFA (pfd)

Put_Free_RFA combines Supply_FD and Supply_RBD algorithms described in "Programming the 82586" into one function. The begin and end pointers delineate what the CPU believes is the beginning and end of the free list. The decision of whether to restart the RU is made when examining both the free FD list and the free RBD list. This is why two ru_start_flags are used, one for the FD list and one for the RBD list. Both flags are initialized to FALSE.

The function starts off by initializing the FD so that the EL bit is set, the status is 0, and the FD link field is NULL. The rbd pointer is saved before the rbd pointer field in the FD is set to NULL. The free FD list is examined and if it's empty, begin—fd and end—fd are loaded with the address of the FD being returned. In this case the RU should not be restarted, because there is only one FD on the free list. If the free FD list is not empty, the FD being returned is placed on the end of the list, the end pointer is updated, and the RU start flag is set TRUE.

To begin the RBD list processing the end of the returned RBD list is determined, and this last RBD's EL bit is set. If the free RBD list is empty, the returned RBD list becomes the free RBD list. If there is more than one RBD on the returned list, the ru start flag is set TRUE. If the free RBD list is not empty, the returned RBD list is appended on the end of the free list, the end—rbd pointer is updated, and the ru start flag is set TRUE.

The last part of Put_Free_RFA( ) is to determine whether to call RU_Start( ). Both ru start flags are ANDed together, and if the result is TRUE, the Ru_Start( ) function is called.

3.5.3 RESTARTING THE RECEIVE UNIT-RU_START()

The Ru_Start( ) function checks two things before it decides to restart the RU. The first thing it checks is whether the RU is already READY. If it is, there is no reason to restart it. If the RU is IDLE or in NO_RESOURCES, then the second thing to check is whether the first free FD on the free FD list has its C bit set. If it does, then the RU should not be restarted. The reason is that the free FD list should only contain free FDs

when the RU is started. If the C bit is set in the FD, then not all the used FD have been removed yet. If the RU is started when used FDs are still in the RFA, the 82586 will write over the used FDs and frames will be lost. Therefore Ru_Start( ) is exited if the first FD in the RFA has its C bit set. If the RU is not READY, and begin_fd doesn't point to a used FD, then the RU is restarted.

Note that in "Programming the 82586" there are two more conditions to be met before the RU is started: two or more FD on the RFA, and two or more RBD on the RFA. These conditions are checked in Put_Free_RFA( ), and Ru_Start( ) isn't called unless they are met.

4.0 LOGICAL LINK CONTROL

The IEEE 802.2 LLC function completes the Data Link Layer of the OSI model. The LLC module in this design example implements a class 1 level of service which provides a connectionless datagram interface. Several data link users or processes can run on top of the data link layer. Each user is identified by a link service access point (LSAP). Communication between data link users is via LSAPs. An LSAP is an address that identifies a specific user process or another layer (see Figure 13). The LSAP addresses are defined as follows:

Data Link Layer (Station Component)	00H
Transport Layer	FEH
Network Management Layer	08H
User Processes	multiples of 4 in the range OCH < LSAP ≤ FCH

Each receiving process is identified by a destination LSAP (DSAP) and each sending process is identified by a source LSAP (SSAP). Before a destination process can receive a packet, its DSAP must be included in a list of active DSAPs for the data link.

Figure 14 illustrates the relationship between the Station Component and the SAP components. (The SAP components are user processes.) The Station Component receives all of the good frames from the Handler and checks the DSAP address. If the DSAP address is 0, then the frame is addressed to the Station Component and a Station Component Response is generated. If the DSAP address is on the active DSAP list, then the Station Component passes the frame to the addressed SAP. If the DSAP address is unknown, the frame is returned to the handler.

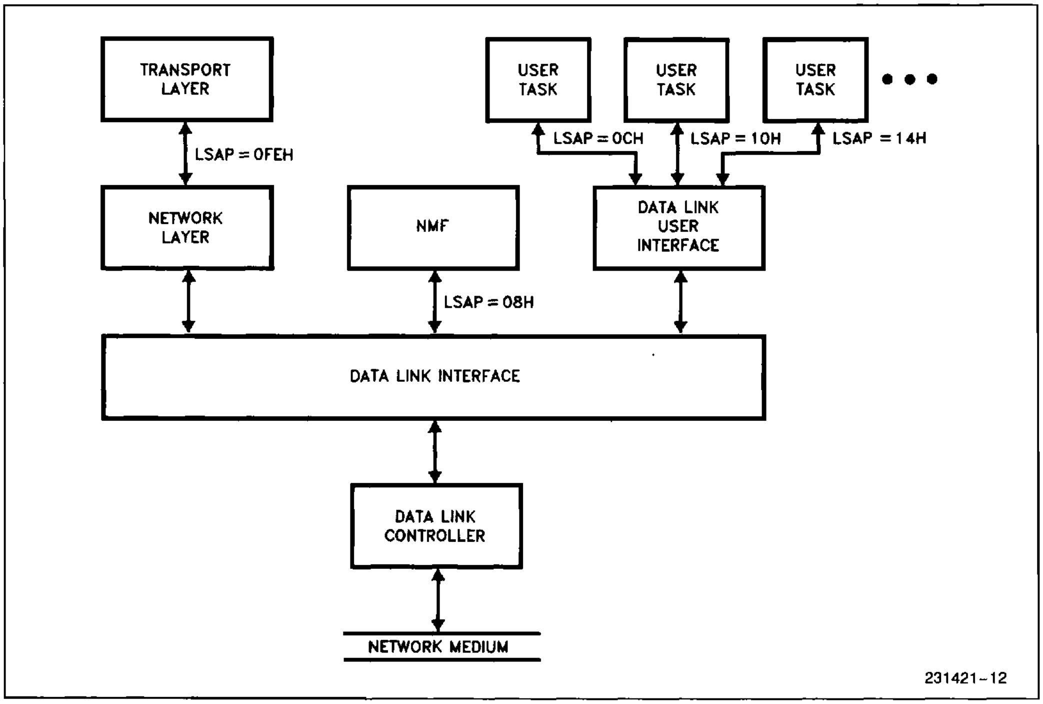

Figure 13. Data Link Interface

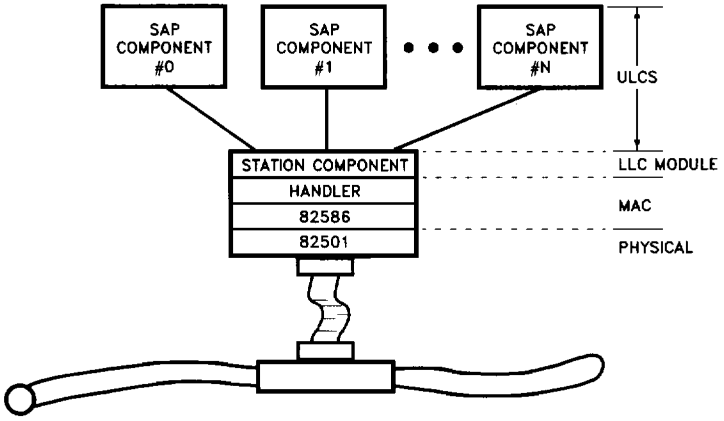

Figure 14. Station Component Relationship

There are 3 commands and 2 responses which the class 1 LLC layer must implement. Figure 15 shows IEEE 802.2 Class 1 commands and responses and Figure 16 shows the IEEE 802.2 Class 1 frame format.

Commands	Responses	Description
UI		Unnumbered Information
XID	XID	Exchange ID
TEST	TEST	Remote Loopback

Figure 15. IEEE 802.2 Class 1, Type 1 Commands and Responses

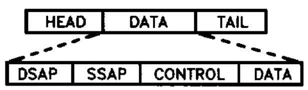

Figure 16. IEEE 802.2 Class 1 Frame Format

From Figure 15 it can be seen that there are no LLC class 1 UI responses because information frames are not acknowledged at the data link level. The only command frames that may require responses are XID and TEST. If a command frame is addressed to the Station Component, it checks the control field to see what type of frame it is. If it's an XID frame, the Station Component responds with a class 1 XID response frame. If it's a TEST frame, the Station Component responds with a TEST frame, echoing back the data it received. In both cases, the response frame is addressed to the source of the command frame.

Any frames addressed to active SAPs are passed directly to them. The Station Component will not respond to SAP addressed frames. Therefore it is the responsibility of the SAPs to recognize and respond to frames addressed to them. When a SAP transmits a frame, it builds the IEEE 802.2 frame itself and calls the Handler's Send_Frame( ) function directly. The LLC module is not used for SAP frame transmission. The only functions which the LLC module implement are the dynamic addition and deletion of DSAPs, multiplexing the frames to user SAPs, and the Station Component command recognition and responses. This is one implementation of the IEEE 802.2 standard. Other implementations may have the LLC module do more functions, such as SAP command recognitions and responses. A list of the functions included in the LLC module is as follows:

LLC Functions	Description
Init_Llc( )	Initializes the DSAP address table and calls Init_586( )
Add_Dsap_ Address (dsap, pfunc)	Add a DSAP address to the active list dsap - DSAP address pfunc - pointer to the SAP function
Delete—Dsap— Address (dsap)	Delete a DSAP address dsap - DSAP address
Recv—Frame (pfd)	Receives a frame from the 82586 Handler pfd - Frame Descriptor Pointer
Station—Component— Response (pfd)	Generates a response to a frame addressed to the Station Component pfd - Frame Descriptor Pointer

4.1 Adding and Deleting LSAPs

When a user process wants to add a LSAP to the active list, the process calls Add__Dsap__Address(dsap, pfunc). The dsap parameter is the actual DSAP address, and the pfunc parameter is the address of the function to be called when a frame with the associated DSAP address is received.

The LLC module maintains a table of active dsaps which consists of an array of structures. Each structure contains two members: stat - indicates whether the address is free or inuse, and (*p__sap__func)( ) contains the address of the function to call. The index into the array of structures is the DSAP address. This speeds up processing by eliminating a linear search. Delete__Dsap__Address (dsap) simply uses the DSAP index to mark the stat field FREE.

5.0 APPLICATION LAYER

For most networks the application layer resides on top of several other layers referred to here as ULCS. These other layers in the OSI model run from the network layer through the presentation layer. The implementation of the ULCS layers is beyond the scope of this application note, however Intel provides these layers as well as the data link layer with the OpenNET product line. For the purpose of this application note the application layer resides on top of the data link layer and its use is to demonstrate, exercise and test the data link layer design example.

There can be several processes sitting on top of the data link layer. Each process appears as a SAP to the data link. The UAP module, which implements the application layer, is the only SAP residing on top of the data link layer in this application example. Other SAPs could certainly be added such as additional "connectionless" terminals, a networking gateway, or a transport layer, however in the interest of time this was not done.

5.1 Application Layer Human Interface

The UAP provides a menu driven human interface via an async terminal connected to port B on the iSBC 186/51 board. The menu of the commands is listed in Figure 17 along with a description that follows:

Terminal Mode - implements a virtual terminal with datagram capability (connectionless "class 1" service). This mode can also be thought of as an async to IEEE 802.2/802.3 protocol converter.

Monitor Mode - allows the station to repeatedly transmit any size frame to the cable. While in the Monitor Mode, the terminal provides a dynamic update of 6 station related parameters.

High Speed Transmit Mode - sends frames to the cable as fast as the software possibly can. This mode demonstrates the throughput performance of the Data Link Driver.

Change Transmit Statistics - When Transmit Statistics is on several transmit statistics are gathered during transmission. If Transmit Statistics is off, statistics are not gathered and the program jumps over the section of code in the interrupt routine which gathers these statistics. The transmission rate is slightly increase when Transmit Statistics is off.

Print All Counters - Provides current information on the following counters.

Good frames transmitted:
Good frames received:

CRC errors received:
Alignment errors received:

Out of Resource frames:
Receiver overrun frames:

Each time a frame has been successfully transmitted the Good frames transmitted count is incremented. The same holds true for reception. CRC, Alignment, Out of Resources, and Overrun Errors are all obtained from the SCB. Underrun, lost CRS, SQE error, Max retry, and Frames that deferred are all transmit statistics that are obtained from the Transmit command status word. 82586 Reset is a count which is incremented each time the 82586 locks up. This count has never normally been incremented.

T - Terminal Mode	M - Monitor Mode
X - High Speed Transmit Mode	V - Change Transmit Statistics
P - Print All Counters	C - Clear All Counters
A - Add a Multicast Address	Z - Delete a Multicast Address
S - Change the SSAP Address	D - Change the DSAP Address
N - Change Destination Node Address	L - Print All Addresses
R - Re-Initialize the Data Link	B - Change the Number Base

Figure 17. Menu of Data Link Driver Commands

Clear All Counters - Resets all of the counters.

Add/Delete Multicast Address - Adds and Deletes Multicast Addresses.

Change SSAP Address - Deletes the previous SSAP and adds a new one to the active list. The SSAP in this case is this stations LSAP. When a frame is received, the DSAP address in the frame received is compared with any active LSAPs on the list. The SSAP is also used in the SSAP field of all transmitted frames.

Change DSAP Address - Delete the old DSAP and add a new one. The DSAP is the address of the LSAP which all transmit frames are sent to.

Change Destination Node Address - Address a new node.

Print All Addresses - Display on the terminal the station address, destination address, SSAP, DSAP, and all multicast addresses.

Re-initialize Data Link - This causes the Data Link to completely reinitialize itself. The 82586 is reset and reinitialized, and the selftest diagnostic and loopback tests are executed. The results of the diagnostics are printed on the terminal. The possible output messages from the 82586 selftest diagnostics are:

Passed Diagnostic Self Tests

Failed: Self Test Diagnose Command

Failed: Internal Loopback Self Test

Failed: External Loopback Self Test

Failed: External Loopback Through Transceiver Self Test

Change Base - Allows all numbers to be displayed in Hex or Decimal.

5.2 A Sample Session

The following text was taken directly from running the Data Link software on a 186/51 board. It begins with the iSDM monitor signing on and continues into executing the Data Link Driver software.

```
iSDM 86 Monitor, V1.0

Copyright 1983 Intel Corporation

.G D000:6

            ****************************************************************
            *                                                              *
            * 82586 IEEE 802.2/802.3 Compatible Data Link Driver *
            *                                                              *
            ****************************************************************

Passed Diagnostic Self Tests

Enter the Address of the Destination Node in Hex -> 00AA0000179E

Enter this Station's LSAP in Hex -> 20

Enter the Destination Node's LSAP in Hex -> 20

Do you want to Load any Multicast Addresses? (Y or N) -> Y

Enter the Multicast Address in Hex -> 00AA00111111

Would you like to add another Multicast Address? (Y or N) -> N

This Station's Host Address is: 00AA00001868

The Address of the Destination Node is: 00AA0000179E

This Station's LSAP Address is: 20

The Address of the Destination LSAP is: 20

The following Multicast Addresses are enabled: 00AA00111111
```

```
Commands are:

T - Terminal Mode                           M - Monitor Mode
X - High Speed Transmit Mode                V - Change Transmit Statistics
P - Print All Counters                      C - Clear All Counters
A - Add a Multicast Address                 Z - Delete a Multicast Address
S - Change the SSAP Address                 D - Change the DSAP Address
N - Change Destination Node Address         L - Print All Addresses
R - Re-Initialize the Data Link             B - Change the number Base

Enter a command, type H for Help -> P

Good frames transmitted:    24     Good frames received:        1
CRC errors received:         0     Alignment errors received:   0
Out of Resource frames:      0     Receiver overrun frames:     0
82586 Reset:                 0     Transmit underrun frames:    0
Lost CRS:                    0     SQE errors:                  9
Maximum retry:               0     Frames that deferred:        4

Enter a command, type H for Help --> T

Would you like the local echo on? (Y or N) --> Y

This program will now enter the terminal mode.

Press ^C then CR to return back to the menu

Hello this is a test.

/*^C CR */

Enter a command, type H for Help --> M

Do you want this station to transmit? (Y or N) --> Y

Enter the number of data bytes in the frame --> 1500

Hit any key to exit Monitor Mode.

 # of Good       # of Good       CRC      Alignment       No        Receive
   Frames          Frames        Errors    Errors        Resource   Overrun
Transmitted       Received                                Errors    Errors

          32              0      00000     00000          00000     00000

/* CR */

Enter a command, type H for Help --> X

Hit any key to exit High Speed Transmit Mode.

/* CR */

Enter a command, type H for Help --> R
Passed Diagnostic Self Tests
```

5.3 Terminal Mode

The Terminal mode buffers characters received from the terminal and sends them in a frame to the cable. When a frame is received from the cable, data is extracted and sent to the terminal. One of three events initiate the UAP to send a frame providing there is data to send: buffering more than 1500 bytes, receiving a Carriage Return from the terminal, or receiving an interrupt from the virtual terminal timer.

The virtual terminal timer employs timer 1 in the 80130 to cause an interrupt every .125 seconds. Each time the interrupt occurs the software checks to see if it received one or more characters from the terminal. If it did, then it sends the characters in a frame.

The interface to the async terminal is a 256 byte software FIFO. Since the terminal communication is full duplex, there are two half duplex FIFOs: a Transmit FIFO and a Receive FIFO. Each FIFO uses two functions for I/O: Fifo__In( ) and Fifo__Out( ). A block diagram is displayed in Figure 18.

The serial I/O for the async terminal interface is always polled except in the Terminal mode where it is interrupt driven. The Terminal mode begins by enabling the 8274 receive interrupt but leaves the 8274 transmit interrupt disabled. This way any characters received from the terminal will cause an interrupt. These characters are then placed in the Transmit FIFO. The only time the 8274 transmit interrupt is enabled is when the Receive FIFO has data in it. The receive FIFO is filled from frames being received from the cable. Each time a transmit interrupt occurs a byte is removed from the Receive FIFO and written to the 8274. When the Receive FIFO empties, the 8274 transmit interrupt is disabled.

The flow control implemented for the terminal interface is via RTS and CTS. When the Transmit FIFO is full, RTS goes inactive preventing further reception of characters (see Table 1). If the Receive FIFO is full, receive frames are lost because there is no way for the data link using class 1 service to communicate to the remote station that the buffers are full. Lost receive frames are accounted for by the Out of Resources Frame counter.

The Async Terminal bit rate sets the throughput capability of the station in the terminal mode because the bottle neck for this network is the RS232 interface. Using this fact a simple test was conducted to verify the data link driver's capability of switching between the receiver's No Resource state and the Ready State. For example if station B is sending frames in the High Speed Transmit mode to station A which is in the Terminal mode, frames will be lost in station A. Under these circumstances station A's receiver will be switching from Ready state to Out of Resources state. The sum of Good frames received plus Out of Resource frames from station A should equal Good frames transmitted from station B; unless there were any underruns or overruns.

Table 1. FIFO State Table

Function	Present State	Next State	Action
FIFO__T__IN( )	EMPTY	IN USE	Start Filling Transmit Buffer
	IN USE	FULL	Shut Off RTS
FIFO__T__OUT( )	FULL	IN USE	Enable RTS
	IN USE	EMPTY	Stop Filling Transmit Buffer
FIFO__R__IN( )	EMPTY	IN USE	Turn on TxInt
	IN USE	FULL	Stop Filling FIFO from Receive Buffer
FIFO__R__OUT( )	FULL	IN USE	Start Filling FIFO from Receive Buffer
	IN USE	EMPTY	Turn Off TxInt

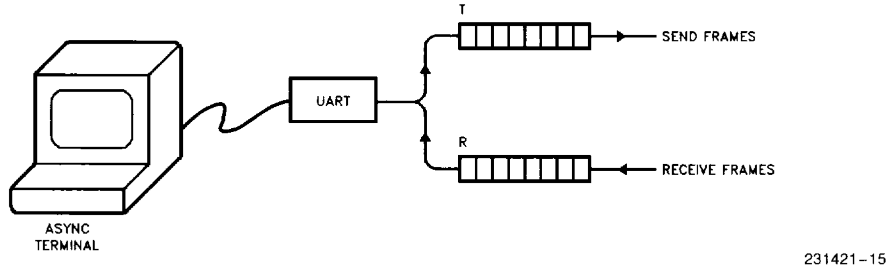

Figure 18

5.3.1 SENDING FRAMES

The Terminal Mode is entered when the Terminal_Mode( ) function is called from the Menu interface. The Terminal_Mode( ) function is one big loop, where each pass sends a frame. Receiving frames in the Terminal Mode is handled on an interrupt driven basis which will be discussed next.

The loop begins by getting a TBD from the 82586 handler. The first three bytes of the first buffer are loaded with the IEEE 802.2 header information. The loop then waits for the Transmit FIFO to become not EMPTY, at which point a byte is removed from the Transmit FIFO and placed in the TBD. After each byte is removed from the Transmit FIFO several conditions are tested to determine whether the frame needs to be transmitted, or whether a new buffer must be obtained. A frame needs to be transmitted if: a Carriage Return is received, the maximum frame length is reached, or the send_frame flag is set by the virtual terminal timer. A new buffer must be obtained if none of the above is true and the max buffer size is reached.

If a frame needs to be sent the last TBD's EOP bit is set and its buffer count is updated. The 82586 Handler's Send_Frame( ) function is called to transmit the frame, and continues to be called until the function returns TRUE.

The loop is repeated until a ^C followed by a Carriage Return is recieved.

5.3.2 RECEIVING FRAMES

Upon initialization the UAP module calls the Add_Dsap_Address(dsap, pfunc) function in the LLC module. This function adds the UAP's LSAP to the active list. The pfunc parameter is the address of the function to call when a frame has been received with the UAP's LSAP address. This function is Recv_Data_1( ). Recv_Data_1( ) looks at the control field of the frame received and determines the action required.

The commands and responses handled by Recv_Data_1( ) are the same as the Station Component's commands and responses given in Figure 15. One difference is that Recv_Data_1( ) will process a UI command while the Station Component will ignore a UI command addressed to it.

Recv_Data_1( ) will discard any UI frames received unless it is in the Terminal Mode. When in the Terminal Mode, Recv_Data_1( ) skips over the IEEE 802.2 header information and uses the length field to determine the number of bytes to place in the Receive FIFO. Before a byte is placed in the FIFO, the FIFO status is checked to make sure it is not full. Recv_Data_1( ) will move all of the data from the frame into the Receive FIFO before returning.

When a frame is received by the 82586 handler an interrupt is generated. While in the 82586 interrupt routine the receive frame is passed to the LLC layer and then to the UAP layer where the data is placed in the Receive FIFO by Recv_Octal_Data_1( ). Since Recv_Data_1( ) will not return until all of the data from the frame has been moved into the Receive FIFO, the 8274 transmit interrupt must be nested at a higher priority than the 82586 interrupt to prevent a software lock. For example if a frame is received which has more than 256 bytes of data, the Receive FIFO will fill up. The only way it can empty is if the 8274 interrupt can nest the 82586 interrupt service routine. If the 8274 could not interrupt the 82586 ISR then the software would be stuck in Recv_Data_1( ) waiting for the FIFO to empty.

5.4 Monitor Mode

The Monitor Mode dynamically updates 6 station related parameters on the terminal as shown below.

The Monitor_Mode( ) function consists of one loop. During each pass through the loop the counters are updated, and a frame is sent. Any size frame can be transmitted up to a size of the maximum number of transmit buffers available. Frame sizes less than the minimum frame length are automatically padded by the 82586 Handler.

The data in the frames transmitted in the Monitor Mode are loaded with all the printable ASCII characters. This way when one station is in the Monitor Mode transmitting to another station in the Terminal Mode, the Terminal Mode station will display a marching pattern of ASCII characters.

# of Good Frames Transmitted	# of Good Frames Received	CRC Errors	Alignment Errors	No Resource Errors	Receive Overrun Errors
32	0	00000	00000	00000	00000

5.5 High Speed Transmit Mode

The High Speed Transmit Mode demonstrates the throughput performance of the 82586 Handler. The Hs_Xmit_Mode( ) function operates in a tight loop which gets a TBD, sets the EOF bit, and calls Send_Frame( ). The flow chart for this loop is shown in Figure 19.

The loop is exited when a character is received from the terminal. Rather than polling the 8274 for a receive buffer full status, the 8274's receive interrupt is used. When the Hs_Xmit_Mode( ) function is entered, the hs_stat flag is set true. If the 8274 receive interrupt occurs, the hs_stat flag is set false. This way the loop only has to test the hs_stat flag rather than calling inb( ) function each pass through the loop to determine whether a character has been received.

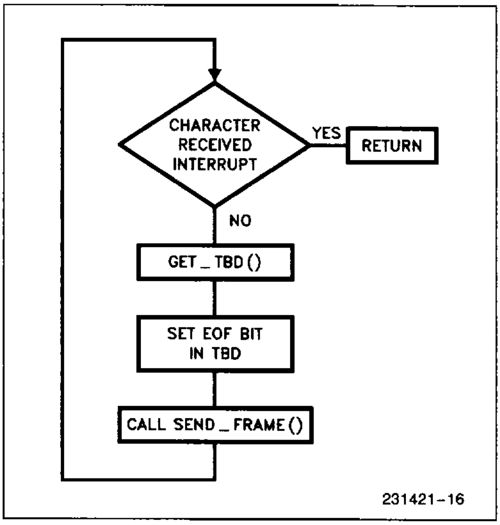

Figure 19. High Speed Transmit Mode Flow Chart

The performance measured on an 8 MHz 186/51 board is 593 frames per second. The bottle neck in the throughput is the software and not the 82586. The size of the buffer is not relevant to the transmit frame rate. Whether the buffer size is 128 bytes or 1500 bytes, linked or not, the frame rate is still the same. Therefore assuming a 1500 byte buffer at 593 frames per second, the effective data rate is 889,500 bytes per second.

This can easily be demonstrated by using two 186/51 boards running the Data Link software. The receiving stations counters should be cleared then placed in the Monitor mode. When placing it in the monitor mode, transmission should not be enabled. When the other station is placed in the High Speed Transmit Mode a timer should be started. One can use a stop watch to determine the time interval for transmission. The frame rate is determined by dividing the number of frames received in the Monitor station by the time interval of transmission.

APPENDIX A
COMPILING, LINKING, LOCATING, AND RUNNING THE SOFTWARE ON THE 186/51 BOARD

********* Instructions for using the 186/51 board *********

Use 27128A for no wait state operation. 27128s can be used but wait states will have to be added.

Copy HI.BYT and LO.BYT files into EPROMs
PROMs go into U34 - HI.BYT and U39 - LO.BYT on the 186/51 board

JUMPERS REQUIRED

Jumper the 186/51 board for 16K byte PROMs in U34 and U39 Table 2-5 in 186/51 HARDWARE REFERENCE MANUAL (Rev-001)

186/51(ES)	186/51 (S)/186/51
E151-E152 OUT	E199-E203 OUT
E152-E150 IN	E203-E191 IN
E94-E95 IN	E120-E119 IN
E100-E106 IN	E116-E112 IN
E107-E113 IN	E111-E107 IN
E133-E134 IN	E94-E93 IN

also change interrupt priority jumpers - switch 8274 and 82586 interrupt priorities

E36-E44 OUT	E43-E47 OUT
E39-E47 OUT	E46-ES0 OUT
E37-E45 OUT	E44-E48 OUT

WIRE WRAP

E36-E47 IN	E43-E50 IN
E39-E44 IN	E46-E47 IN
E79-E45 IN	E90-E48 IN

USE SDM MONITOR

The SDM Monitor should have the 82586's SCP burned into ROM. The ISCP is located at OFFFOH. Therefore for the SCP the value in the SDM ROM should be:

ADDRESS	DATA
FFFF6H	XXOOH
FFFF8H	XXXXH
FFFFAH	XXXXH
FFFFCH	FFFOH
FFFFEH	XXOOH

To run the program begin execution at 0D000:6H

```
I.E. G D000:6

GOOD LUCK!

**********    submit file for compiling one module:    **********

run

cc86.86  :F6:%0 LARGE ROM DEBUG DEFINE(DEBUG) include(:F6:)

exit

**********    submit file for linking and locating:    **********

run

link86     :F6:assy.obj, :F6:dld.obj, :F6:llc.obj, &

:F6:uap.obj, lclib.lib to :F6:dld.lnk segsize(stack(4000h)) notype

loc86 :F6:dld.lnk to :F6:dld.loc&

initcode (0D0000H) start(begin) order(classes(data, stack, code)) &
addresses(classes(data(3000H), stack(0CB00H), code(0D0020H)))

oh86 :F6:dld.loc to :F6:dld.rom

exit

**********    submit file for burning EPROMs using IPPS:    **********

ipps

i 86

f :F6:dld.rom (0d0000h)

3

2

1

0 to :F6:lo.byt

y

1 to :F6:hi.byt

y


t 27128

9

c :F6:lo.byt t p

n

C :F6:hi.byt t p

n

exit
```

```
/PCO/USR/CHUCK/CSRC/DLD.H

/************************************************************************
*                                                                       *
*                     82586 Structures and Constants                    *
*                                                                       *
************************************************************************/

/* general purpose constants */

#define INUSE   0
#define EMPTY   1
#define FULL    2
#define FREE    1
#define TRUE    1
#define FALSE   0
#define NULL    0xFFFF

/* Define Data Structures */

#define RBUF_SIZE       128 /* receive buffer size */
#define TBUF_SIZE       128 /* transmit buffer size */
#define ADD_LEN         6
#define MULTI_ADDR_CNT  16

typedef unsigned short int u_short;

/* results from Test_Link(): loaded into Self_Test char */

#define PASSED                     0
#define FAILED_DIAGNOSE            1
#define FAILED_LPBK_INTERNAL       2
#define FAILED_LPBK_EXTERNAL       3
#define FAILED_LPBK_TRANSCEIVER 4

/* Frame Commands */
#define      UI       0x03     /* Unnumbered Information Frame */
#define      XID      0xAF     /* Exchange Identification */
#define      TEST     0xE3     /* Remote Loopback Test */
#define      P_F_BIT 0x10      /* Poll/Final Bit Position */
#define      C_R_BIT 0x01      /* Command/Response bit in SSAP */


#define      DSAP_CNT    8     /* Number of allowable DSAPs; must be a multiple
                                  of 2**N, and DSAP addresses assigned must be
                                  divisible by 2**(8-N).
                                  (i.e. the N LSBs must be 0) */

#define      DSAP_SHIFT  5     /* DSAP_SHIFTS must equal 8-N */


#define      XID_LENGTH  6     /* Number of Info bytes for XID Response frame */


/* System Configuration Pointer SCP */

   struct SCP  {
                 u_short sysbus; /* 82586 bus width, 0 - 16 bits
                                    1 - 8 bits */
```

231421-17

```
/PCO/USR/CHUCK/CSRC/DLD.H

                u_short junk[2];
                u_short iscpl;  /* lower 16 bits of iscp address */
                u_short iscph;  /* upper 8 bits of iscp address */
               };

/* Intermediate System Configuration Pointer ISCP */

   struct ISCP {
                u_short busy ;  /*set to 1 by cpu before its first CA,
                               cleared by 82586 after reading */
                u_short offset  ;   /* offset of system control block */
                u_short base1 ; /* base of system control block */
                u_short base2 ;
               } ;

/* System Control Block SCB */

   struct SCB  {
                u_short stat;       /* Status word */
                u_short cmd;        /* Command word */
                u_short cbl_offset; /* Offset of first command block in CBL */
                u_short rfa_offset; /* Offset of first frame descriptor in RFA */
                u_short crc_errs;   /* CRC errors accumulated */
                u_short aln_errs;   /* Alignment errors */
                u_short rsc_errs;   /* Frames lost because of no Resources */
                u_short ovr_errs;   /* Overrun errors */
               };

/*  Command Block */

   struct  CB  {
                u_short stat;   /* Status of Command */
                u_short cmd;    /* Command */
                u_short link;   /* link field */
                u_short parm1;  /* Parameters */
                u_short parm2;
                u_short parm3;
                u_short parm4;
                u_short parm5;
                u_short parm6;
           };


/*  Multicast Address Command Block MA_CB */

   struct MA_CB{
                u_short stat;   /* Status of Command */
                u_short cmd;    /* Command */
                u_short link;   /* Link field */
                u_short mc_cnt; /* Number of MC addresses */
                char mc_addr[ADD_LEN*MULTI_ADDR_CNT]; /* MC address area */
               };


/* Transmit Buffer Descriptor TBD */

   struct TBD  {
```

231421-18

/PCO/USR/CHUCK/CSRC/DLD.H

```
                u_short act_cnt;    /* Number of bytes in buffer */
                u_short link;       /* offset to next TBD */
                u_short buff_l;     /* lower 16 bits of buffer address */
                u_short buff_h;     /* upper 8 bits of buffer address */
                struct TB *buff_ptr;    /* not used by the 586: used by the
                                   software to save address translation
                                   routine.  */
               };

/* Transmit Buffers */
    struct TB   {
                 char data [TBUF_SIZE];
                };

/* Frame Descriptor FD */

   struct  FD  {
                u_short stat;               /* Status Word of FD */
                u_short el_s;               /* EL and S bits */
                u_short link;               /* link to next FD */
                u_short rbd_offset;         /* Receive buffer descriptor offset */
                char dest_addr[ADD_LEN]; /*Destination address */
                char src_addr[ADD_LEN]; /* Source address */
                u_short length;             /* Length field */
               };

/* Receive Buffer Descriptor RBD */

   struct RBD {
                u_short act_cnt;        /* Actual number of bytes received */
                u_short link;           /* Offset to next RBD */
                u_short buff_l;         /* Lower 16 bits of buffer address */
                u_short buff_h;         /* upper 8 bits of buffer address */
                u_short size;           /* size of buffer */
                struct RB *buff_ptr;    /* not used by the 586: used by the
                                       software to save address translation
                                       routine. */
               };


/* Receive Buffers */

   struct RB   {
                char data[RBUF_SIZE];
               };


struct  FRAME_STRUCT
        {
            unsigned char   dsap;   /* Destination Service Access Point */
            unsigned char   ssap;   /* Source Service Access Point */
            unsigned char   cmd;    /* ISO Data Link Command */
        };

    /* LSAP Address Table */
struct  LAT {
                char    stat,           /* INUSE or FREE */
```

231421-19

```
/PCO/USR/CHUCK/CSRC/DLD.H

                int     (*p_sap_func)();/* Pointer to LSAP function; associated
                                           with dsap address */
           };


struct MAT {                            /* Multicast Address Table */
        char    stat;                   /* INUSE or FREE */
        char    addr[ADD_LEN];          /* actual mc address */
        };

/* general purpose flags */

struct FLAGS {
    unsigned diag_done : 1 ;            /* diagnose command complete */
    unsigned stat_on  : 1 ;             /* network diagnostic statistics on/off */
    unsigned reset_sema: 1 ;            /* don't reset when this bit is set */
    unsigned reset_pend: 1 ;            /* reset when this bit is set */
    unsigned lpbk_test: 1 ;             /* loopback test flag */
    unsigned lpbk_mode: 1 ;             /* loopback mode on/off */
       } ;


/* General purpose bits */

#define ELBIT   0x8000
#define EOFBIT  0x8000
#define SBIT    0x4000
#define IBIT    0x2000
#define CBIT    0x8000
#define BBIT    0x4000
#define OKBIT   0x2000

/* SCB patterns */

#define CX          0x8000
#define FR          0x4000
#define CNA         0x2000
#define RNR         0x1000
#define RESET       0x0080
#define CU_START    0x0100
#define RU_START    0x0010
#define RU_ABORT    0x0040
#define CU_MASK     0x0700
#define RU_MASK     0x0070
#define RU_READY    0x0040

/* 82586 Commands */

#define NOP         0x0000
#define IA          0x0001
#define CONFIGURE   0x0002
#define MC_SETUP    0x0003
#define TRANSMIT    0x0004
#define TDR         0x0005
#define DUMP        0x0006
#define DIAGNOSE    0x0007
```

231421-20

```
/PCD/USR/CHUCK/CSRC/DLD.H

/* 82586 Command and Status Masks */

#define CMD_MASK          0x0007
#define NOERRBIT          0x2000
#define COLLMASK          0x000F
#define DEFERMASK         0x0080
#define NOCRSMASK         0x0400
#define UNDERRUNMASK      0x0100
#define SQEMASK           0x0040
#define MAXCOLMASK        0x0020
#define OUT_OF_RESOURCES 0x0200

/* Configure Parameters */

#define FIFO_LIM     0x0800       /* use FIFO lim of 8 */
#define BYTE_CNT     0x000B
#define SRDY         0x0040
#define SAV_BF       0x0080
#define ADDR_LEN     0x0600       /* address length of 6 bytes */
#define AC_LOC       0x0800
#define PREAM_LEN    0x2000       /* preamble length of 8 bytes */
#define INT_LPBCK    0x4000
#define EXT_LPBCK    0x8000
#define LIN_PRIO     0x0000       /* no priority */
#define ACR          0x0000
#define BOF_MET      0x0080
#define IFS          0x6000       /* IFS time 9.6 usec */
#define SLOT_TIME    0x0200       /* slot time 51.2 usec */
#define RETRY_NUM    0xF000       /* retry number 15 */
#define PRM          0x0001
#define BC_DIS       0x0002
#define MANCHESTER   0x0004
#define TONO_CRS     0x0008
#define NCRC_INS     0x0010
#define CRC_16       0x0020
#define BT_STUFF     0x0040
#define PAD          0x0080
#define CRSF         0x0000       /* no carrier sense filter */
#define CRS_SRC      0x0800
#define CDTF         0x0000       /* no collision detect filter */
#define CDT_SRC      0x8000
#define MIN_FRM_LEN 0x0040        /* 64 bytes */
#define MIN_DATA_LEN MIN_FRM_LEN - 18   /* assumes Ethernet/IEEE 802.3
                                           frames with 6 bytes of address */
#define MAX_FRAME_SIZE  1500 - 3
```

231421-21

1

```
/PCO/USR/CHUCK/CBRC/DLD.C

/**************************************************************************
*                                                                         *
*                              82586 Handler                              *
*                                                                         *
**************************************************************************/


/* Define constants for storage area */

#define CB_CNT      8   /* Number of available Command Blocks */
#define FD_CNT      16  /* Number of available Frame Descriptors */
#define RBD_CNT     64  /* Number of available Receive Buffer descriptors */
#define TBD_CNT     16  /* Number of available Transmit Buffer descriptors */

/* loopback parameters passed to Configure() */

#define INTERNAL_LOOPBACK   0x4000
#define EXTERNAL_LOOPBACK   0x8000
#define NO_LOOPBACK         0x0000

#include "dld.h"            /* 586 Data Structures */


/* 186 Timer Addresses */

#define TIMER1_CTL  0xFF5E
#define TIMER1_CNT  0xFF58
#define TIMER2_CTL  0xFF66
#define TIMER2_CNT  0xFF60


/* external functions */

/* I/O */
int     inw();      /* input word : inw(address) */
void    outw();     /* output word: outw(address, value) */
void    init_intv();/* initialize the interrupt vector table */
void    enable();   /* enable 80186 interrupts */
void    disable();  /* disable 80186 interrupts */

extern  char    *Build_Ptr();

u_short     SEGMT;      /* Data segment value */
char        *pNULL;     /* NULL pointer */

/* Macro `type' of definitions */

#define CA  outw(0xC8,0)    /* the command to issue a Channel Attention */

#define ESI_LOOPBACK outw(0xC8,0) /* put the ESI in Loopback */
#define NO_ESI_LOOPBACK outw(0xC8,8) /* take the ESI out of Loopback */

#define EOI_80130          outb(0xE0,0x63)    /* End Of Interrupt */
#define TIMER1_EOI_80186 outw(0xFF22,0x04) /* EOI for Timer 1 on the 186 */
#define TIMER1_EOI_80130 outb(0xE0,0x64) /*EOI for 186's Timer1 on the 130 */
```

231421-22

```
/PCO/USR/CHUCK/CSRC/DLD.C

/******** memory  allocation ****************/

int Self_Test;         /* used for diagnostic purposes */
u_short temp;          /* temporary storage */

#define LPBK_FRAME_SIZE     4              /* loopback frame storage */
char     lpbk_frame[LPBK_FRAME_SIZE] = {
0x55, 0xAA, 0x55, 0xAA};

#define whoami_io_add   0x00F0  /* I/O address of Host Address Prom */
char     whoami[ADD_LEN];       /* Ram array where host address is stored */


/* transmission statistic variables */

unsigned long   good_xmit_cnt;
u_short         underrun_cnt;
u_short         no_crs_cnt;
unsigned long   defer_cnt;
u_short         sqe_err_cnt;
u_short         max_col_cnt;
unsigned long   recv_frame_cnt;
u_short         reset_cnt;

    /* Allocate storage for structures and buffers */


struct FLAGS flags;


/* 586 structures */

/* System Configuration Pointer: Rom Initialization */
/* struct SCP scp = {0x0000,0x0000,0x0000,0x1FF6,0x0000};   */

/* struct ISCP iscp;     Intermediate System Configuration Pointer */

struct SCB scb;      /* System Control Block */

struct CB cb[CB_CNT],    /* Command Blocks */
*cb_tos, *begin_cbl, *end_cbl;
                              /* pointer to the beginning of the free
                                 command block list (cb_tos) and the
                                 beginning and end of the 82586 cbl */

struct TBD tbd[TBD_CNT],    /* Transmit Buffer Descriptor */
*tbd_tos;                /* pointer to the free Transmit buffer
                                 descriptors */

struct TB tbuf[TBD_CNT];     /* Transmit Buffers */


struct FD fd[FD_CNT],        /* Frame Descriptors */
*begin_fd, *end_fd;         /* pointers to the beginning and end of
                                  the free FD list */

struct RBD rbd[RBD_CNT],     /* Receive Buffer Descriptors */
```

231421-23

```
/PCO/USR/CHUCK/CSRC/DLD.C

*begin_rbd, *end_rbd;          /* pointers to the beginning and the
                                  end of the rbd list */

struct RB rbuf[RBD_CNT];      /* Receive Buffers */

struct MAT mat[MULTI_ADDR_CNT];       /* Multicast Address Table */
struct MA_CB ma_cb;                   /* Multicast Address Command Block */


/* The following structures are used only in Reset_586() function */
struct      CB       res_cb;      /* Temporary CB for reinitializing the 586 */
struct      MA_CB    res_ma_cb;   /* Temporary MA_CB for reloading Multicast */

/*  Hardware Support Functions */

Enable_586_Int()
{
    int      c;

    c = inb(0xE2);              /* read the 80130 interrupt mask register */
    outb(0xE2, 0x00F7 & c);  /* write to the 80130 interrupt mask register */
}

Disable_586_Int()
{
    int      c;

    c = inb(0xE2);
    outb(0xE2, 0x0008 | c);
}

Set_Timeout()
{
    outw(TIMER1_CNT, 0);     /* Write a 0 to Timer1 count register */
    outw(0xFF5E, 0xE009);    /* Set ENable bit in Timer1 Mode/Control register */
}

Reset_Timeout()
{
    outw(0xFF5E, 0x6009);  /* Reset ENable bit in Timer1 Mode/Control register */
}

Init_Timer() /* 186's Timer 2 is a prescaler for Timer 1. It clocks Timer 1
                every 32.7 msec. The deadman timeout is set for 1.25 sec */
{
    outw(0xFF38, 0x000C);     /* Set Timer1 Interrupt Control register */
    outw(0xFF62, 0xFFFF);     /* set max count register for timer2 to 0FFFFH */
    outw(0xFF5A,      38);    /* set max count register A for timer 1 */
    outw(0xFF66, 0xC001);     /* Set Timer2 Mode/Control register */
    outw(0xFF5E, 0x6009);     /* Set Timer1 Mode/Control register */
    outw(0xFF28, (inw(0xFF28) & 0xFFEF)); /* Enable 186 Timer1 interrupt */
    outb(0xE2,(inb(0xE2) & 0x00EF));  /* enable 80130 interrupt from 80186 */
}

/* end hardware support functions */

Clear_Cnt()
```

231421-24

/PCD/USR/CHUCK/CSRC/DLD.C

```
{
    scb.crc_errs = 0;          /* clear 586 error statistic counters */
    scb.aln_errs = 0;
    scb.rsc_errs = 0;
    scb.ovr_errs = 0;

    good_xmit_cnt = 0;         /* init data link statistics */
    underrun_cnt = 0;
    no_crs_cnt = 0;
    defer_cnt = 0;
    sqe_err_cnt = 0;
    max_col_cnt = 0;
    recv_frame_cnt = 0;
    reset_cnt = 0;
}

Init_586()
{
    struct ISCP *piscp;
    u_short i;
    struct  MAT *pmat;

    NO_ESI_LOOPBACK;  /* Done for 82501. Inactivates CRS if powered up
                         in loopback */
    ESI_LOOPBACK;

    init_intv();     /* Initialization DLDs interrupt vectors */

    Init_Timer();

    flags.reset_sema = 0;    /* Initialize Reset Flags */
    flags.reset_pend = 0;
    flags.stat_on = 1;

    Disable_586_Int();


    piscp =  0x0000FFF0 ; /* Initialize the ISCP pointer*/
    piscp->busy = 1;
    piscp->offset =  Offset(&scb);
    piscp->base1 = SEGMT << 4;
    piscp->base2 = (SEGMT >> 12) & 0x000F ;

    pNULL = Build_Ptr(NULL);     /* build a NULL pointer - 8086 type: 32 bits */
    Build_Rfa();     /* init Receive Frame Area */
    Build_Cb(); /* init Command Block list */
    mc_cb.cmd = 0;        /* multicast address semaphore init */

    Clear_Cnt();

    scb.stat = 0;

    CA;      /* wait for the 586 to complete initialization */

    for ( i = 0; i <= 0xFF00; i++)
```

231421-25

```
/PCO/USR/CHUCK/CSRC/DLD.C

        if (scb.stat == (CX | CNA))
            break;

    if (i >0xFF00)
    Fatal("DLD:init - Did not get an interrupt after Reset/CA\n");

    /* Ack the reset Interrupt */
    scb.cmd = (CX | CNA);
    CA;
    Wait_Scb();
    Enable_586_Int();

    scb.cbl_offset = Offset(&cb[0]);     /* link scb to cb and fd lists */
    scb.rfa_offset = Offset(&fd[0]);

     /* move the prom bytes into whoami array */

    for (i = 0; i < ADD_LEN; i++)
        whoami[(ADD_LEN - 1) - i] = inb(whoami_io_add + i*2);

        /* Initialization the Multicast Address Table */

    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        pmat->stat = FREE;

    Configure(INTERNAL_LOOPBACK);    /* Put 586 in internal loopback */

    SetAddress();    /* Set up the station address */

    /* run diagnostics */

    Test_Link();

    if (Self_Test != PASSED)
        return(Self_Test);

    Configure(NO_LOOPBACK); /* Configure the 82586 */

    return(Self_Test);
}

Build_Rfa()
{
    struct   FD      *pfd;
    struct   RBD     *prbd;
    struct   RB      *pbuf;
    unsigned long    badd;

    /* Build a linear linked frame descriptor list */

    for (pfd = &fd[0]; pfd <= &fd[FD_CNT - 1]; pfd++) {

        pfd->stat = pfd->el_s = 0;
        pfd->link = Offset(pfd+1);
        pfd->rbd_offset = NULL;
    }
```

231421-26

1

```
/PCO/USR/CHUCK/CBRC/DLD.C

    end_fd = --pfd;                 /* point to &fd[FD_CNT - 1] */
    pfd->link = NULL;               /* last fd link is NULL */
    pfd->el_s = ELBIT;              /* last fd has EL bit set */
    begin_fd = pfd = &fd[0];        /* point to first fd */
    pfd->rbd_offset = Offset(&rbd[0]);  /* link first fd to first rbd */

    /* Build a linear linked receive buffer descriptor list */

    for (prbd = &rbd[0], pbuf = &rbuf[0]; prbd <= &rbd[RBD_CNT - 1];
                                                         prbd++, pbuf++) {
        badd = SEGMT << 4;
        badd += Offset(pbuf);
        prbd->buff_l = badd;
        prbd->buff_h = badd >> 16;
        prbd->buff_ptr = pbuf;

        prbd->act_cnt =  0;
        prbd->link = Offset(prbd + 1);
        prbd->size = RBUF_SIZE;
    }

    end_rbd = --prbd;
    prbd->link = NULL;          /* last rbd points to NULL */
    prbd->size |= ELBIT;        /* last rbd has el bit set */

    begin_rbd = &rbd[0];

}

Build_Cb()  /* Build a stack of free command blocks */
{
    struct  CB  *pcb;
    struct  TBD *ptbd;
    struct  TB  *pbuf;
    unsigned    long    badd;

    for (pcb = &cb[0]; pcb <= &cb[CB_CNT - 1]; pcb++) {
        pcb->stat = 0;
        pcb->cmd = ELBIT;
        pcb->link = Offset(pcb + 1);
    }
    --pcb;
    begin_cbl = end_cbl = pNULL;
    pcb->link = NULL;
    cb_tos = &cb[0];

    /* Build a stack of transmit buffer descriptors */

    for (ptbd = &tbd[0], pbuf = &tbuf[0]; ptbd <= &tbd[TBD_CNT - 1];
                                                          ptbd++, pbuf++) {

        ptbd->act_cnt  = TBUF_SIZE;
        ptbd->link = Offset(ptbd + 1);

        badd = SEGMT << 4;
```

231421-27

/PCD/USR/CHUCK/CSRC/DLD.C

```
        badd += Offset(pbuf);
        ptbd->buff_l = badd;
        ptbd->buff_h = badd >> 16;
        ptbd->buff_ptr = pbuf;
    }

    --ptbd;
    ptbd->link = NULL;  /* last tbd link is NULL */
    tbd_tos = &tbd[0];  /* Set the Top Of the Stack */

}

    /* Get a Command Block from the free list */

struct  CB   *Get_Cb() /* return a pointer to a free command block */
{
    struct  CB *pcb;

    if (Offset(pcb = cb_tos) == NULL)
        return(pNULL);
    cb_tos = (struct CB *) Build_Ptr(pcb->link);
    pcb->link = NULL;
    return(pcb);
}

/* Put a Command Block back onto the free list */

Put_Cb(pcb)

    struct  CB *pcb;

{
    pcb->stat = 0;
    pcb->link = Offset(cb_tos);
    cb_tos = pcb;
}

struct TBD  *Get_Tbd()  /* return a pointer to a free transmit buffer
                           descriptor */
{
    struct   TBD    *ptbd;

    flags.reset_sema = 1;
    Disable_586_Int();
    if ((ptbd = tbd_tos) != pNULL) {
        tbd_tos = (struct TBD *) Build_Ptr(ptbd->link);
        ptbd->link = NULL;
    }
    Enable_586_Int();
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(ptbd);
}


Put_Tbd(ptbd)
```

231421-28

```
/PCO/USR/CHUCK/CSRC/DLD.C


struct      TBD     *ptbd;
{
    struct      TBD     *p ;

    /* find the end of the tbd list returned, ptbd is the beginning */

    for (p = ptbd; p->link != NULL; p = (struct TBD *) Build_Ptr(p->link)) ;

    p->act_cnt = TBUF_SIZE;      /* clear EOFBIT and update size on last tbd */
    p->link = Offset(tbd_tos);
    tbd_tos = ptbd;
}


SetAddress()
{
    struct  CB  *pcb;

#ifdef DEBUG

    if ((pcb = Get_Cb()) == pNULL)
    Fatal("dld.c - SetAddress - couldn't get a CB\n");

#else

    pcb = Get_Cb();

#endif  /* DEBUG */

    bcopy((char *)&pcb->parm1, &whoami[0], ADD_LEN); /* move the prom
                                                         address to IA cmd */
    pcb->cmd = IA | ELBIT;

    Issue_CU_Cmd(pcb);
}


Wait_Scb()  /* wait for the scb command word to be clear */
{
    u_short     i, stat;

    for (stat = FALSE; stat == FALSE; ) {

        for (i=0; i<=0xFF00; i++)
            if (scb.cmd == 0)
                break;

        if (i > 0xFF00) {
            Bug("DLD: Scb command not clear\n");
            CA;
        }
        else
            stat = TRUE;
    }
```

231421-29

```
/PCO/USR/CHUCK/CSRC/DLD.C

}

Issue_CU_Cmd(pcb) /* Queue up a command and issue a start CU command if no
                     other commands are queued */
    struct  CB  *pcb;
{
    Disable_586_Int();
    if (begin_cbl == pNULL) {   /* if the list is inactive start CU */
        begin_cbl = end_cbl = pcb;
        scb.cbl_offset = Offset(pcb);
        Wait_Scb();
        scb.cmd = CU_START;
        Set_Timeout();              /* set deadman timer for CU */
        CA;
    }
    else {
        end_cbl->link = Offset(pcb);
        end_cbl = pcb;
    }
    Enable_586_Int();
}

Isr7()
{
    outb(0xE0, 0x67);    /* EOI 80130 */
}

Isr6()
{
    Write("\nInterrupt 6\n");
    outb(0xE0, 0x66);    /* EOI 80130 */
}

Isr5()
{
    Write("\nInterrupt 5\n");
    outb(0xE0, 0x65);    /* EOI 80130 */
}

    /* Deadman Timer Interrupt Service Routine */

Isr_Timeout()    /* Interrupt 4 */
{
    Reset_Timeout();
    if (flags.reset_sema == 1)
        flags.reset_pend = 1;
    else
        Reset_586();

    TIMER1_EOI_80186;
    TIMER1_EOI_80130;
}

    /* Interrupt 0 is Uart in UAP Module */
    /* Interrupt 2 is Timer in UAP Module */
```

231421-30

```
/PCO/USR/CHUCK/CSRC/DLD.C

Isr1()
{
    Write("\nInterrupt 1\n");
    outb(0xE0, 0x61);    /* EOI 80130 */
}

    /* 586 Interrupt service routine: Interrupt 3 */

Isr_586()
{

    u_short     stat_scb;
    struct  CB      *pcb;

    enable();    /* nesting only the uart interrupt */

    Wait_Scb();
    scb.cmd = (stat_scb = scb.stat) & (CX | CNA | FR | RNR);
    CA;

    if (stat_scb & (FR | RNR))
        Recv_Int_Processing();

    if (stat_scb & CNA) {  /* end of cb processing */

        Reset_Timeout();    /* clear deadman timer  */
        pcb = Build_Ptr(scb.cbl_offset);

#ifdef DEBUG

        if (begin_cbl == pNULL){
            Bug("DLD: begin_cbl == NULL in interrupt routine\n");
            return;
        }

        if ((pcb->stat & 0xC000) != 0x8000)
            Fatal("DLD: C bit not set or B bit set in interrupt routine\n");

#endif /* DEBUG */

        switch (pcb->cmd & CMD_MASK) {

        case TRANSMIT:

            if (flags.stat_on == 1) {/* if Transmit Statistics are collected do */

                /* if sqe bit = 0 and there were no collisions -> sqe error
                   this condition will occur on the first transmission if
                   there were no collisions, or if the previous transmit
                   command reached the max collision count, and the current
                   transmission had no collisions */

                if ((pcb->stat & (SQEMASK | MAXCOLMASK | COLLMASK)) == 0)
                    ++sqe_err_cnt;

                if (pcb->stat & DEFERMASK)
                    ++defer_cnt;
```

231421-31

```
/PCO/USR/CHUCK/CSRC/DLD.C


                if (pcb->stat & NOERRBIT)
                    ++good_xmit_cnt;
                else {

                    if (pcb->stat & NOCRSMASK)
                        ++no_crs_cnt;
                    if (pcb->stat & UNDERRUNMASK)
                        ++underrun_cnt;
                    if (pcb->stat & MAXCOLMASK)
                        ++max_col_cnt;
                }
            }
            if (pcb->parm1 != NULL)
                Put_Tbd(Build_Ptr(pcb->parm1));
            break;

        case DIAGNOSE:

            flags.diag_done = 1;
            if ((pcb->stat & NOERRBIT) == 0)
                Self_Test = FAILED_DIAGNOSE;
            break;


        default:
                ;
        }


        /* check to see if another command is queued */

        if (pcb->link == NULL)
            begin_cbl = pNULL;

        else { /* restart the CU and execute the next command on the cbl */

            begin_cbl = Build_Ptr(pcb->link);
            scb.cbl_offset = pcb->link;
            Wait_Scb();
            scb.cmd = CU_START;
            CA;
            Wait_Scb();
            Set_Timeout();        /* START deadman timer */
        }
        if ((pcb->cmd & CMD_MASK) == MC_SETUP)
            pcb->cmd = 0; /* clear MC_SETUP cmd word, this will implement a
                             lock semephore so that it won't be reused until
                             it is completed */
        else
            Put_Cb(pcb); /* Don't return MC_SETUP cmd block. It's not a
                            general purpose command block from free CB list */
    }
    disable();  /* disable cpu int so that the 586 isr will not nest */
    EOI_80130;
}
```

231421-32

```
/PCO/USR/CHUCK/CSRC/DLD.C


Recv_Int_Processing()
{
    struct      FD      *pfd;   /* points to the Frame Descriptor */
    struct      RBD     *q,     /* points to the last rbd for the frame */
                        *prbd;  /* points to the first rbd for the frame */

    for (pfd = begin_fd; pfd != pNULL; pfd = begin_fd)
        if (pfd->stat & CBIT) {
            begin_fd = (struct FD *) Build_Ptr(pfd->link);
            prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
            if (prbd != pNULL) { /* check to see if a buffer is attached */

#ifdef DEBUG
                if (prbd != begin_rbd)
                    Fatal("DLD: prbd != begin_rbd in Recv_Int_Processing\n");
#endif /* DEBUG */
                for (q = prbd; (q->act_cnt & EOFBIT) != EOFBIT;
                                        q = (struct RBD *) Build_Ptr(q->link));

                begin_rbd = (struct RBD *) Build_Ptr(q->link);
                q->link = NULL;
            }
            if (pfd->stat & OUT_OF_RESOURCES)
                Put_Free_RFA(pfd);
            else {
                /* if the DLD is in a loopback test, check the frame recv */
                if (flags.lpbk_mode == 1)
                    Loopback_Check(pfd);
                else

            /* if it's a multicast address check to see if it's
                   in the multicast address table, if not discard the frame */

              if ( ((pfd->dest_addr[0] & 01) == 01) && (!Check_Multicast(pfd)))
                    Put_Free_RFA(pfd);
              else
                {   Recv_Frame(pfd);
                    ++recv_frame_cnt;
                }
            }
        }
        else {
            Ru_Start(); /* If RU has gone into no resources, restart it */
            break;
        }
}


Loopback_Check(pfd)         /* Called by Recv_Int_Processing; checks address
                                   and data of potential loopback frame */
    struct  FD  *pfd;
{
    struct RBD  *prbd;
    struct RB   *pbuf;
```

231421-33

```
/PCO/USR/CHUCK/CSRC/DLD.C

    if ( bcmp((char *) &pfd->src_addr[0], &whoami[0], ADD_LEN) != 0 ) {
        Put_Free_RFA(pfd);
        return;
    }
    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset); /* point to receive
                                                          buffer descriptor */
    pbuf = (struct RB *) prbd->buff_ptr; /* point to receive buffer */

    if ( bcmp((char *) pbuf, &lpbk_frame[0], LPBK_FRAME_SIZE) != 0) {
        Put_Free_RFA(pfd);
        return;
    }

    flags.lpbk_test = 1;  /* passed loopback test */
    Put_Free_RFA(pfd);
}

Check_Multicast(pfd)     /* returns true if multicast address is in MAT */
    struct      FD  *pfd;
{
    struct      MAT *pmat;

    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE &&
        (bcmp((char *) &pfd->dest_addr[0], &pmat->addr[0], ADD_LEN) == 0))
        break;

    if (pmat > &mat[MULTI_ADDR_CNT - 1])
        return(FALSE);
    return(TRUE);
}


    /* Test the Link function: executes Diagnose and Loopback tests */

Test_Link()
{
    Self_Test = PASSED;
    Diagnose();
    if (Self_Test == FAILED_DIAGNOSE)
        return;
    Ru_Start();       /* start up the RU for loopback tests */
    flags.lpbk_mode = 1; /* go into loopback mode */

    flags.lpbk_test = 0; /* set looback test to false */
    Send_Lpbk_Frame();  /* internal loopback test */
    if (flags.lpbk_test == 0) {
        Self_Test = FAILED_LPBK_INTERNAL;
        flags.lpbk_mode = 0;
        return;
    }

    flags.lpbk_test = 0;
    Configure(EXTERNAL_LOOPBACK);    /* external loopback test w/ ESI in lpbk */
    Send_Lpbk_Frame();
    if (flags.lpbk_test == 0) {
        Self_Test = FAILED_LPBK_EXTERNAL;
```

231421-34

1

```
/PCD/USR/CHUCK/CSRC/DLD.C

        flags.lpbk_mode = 0;
        return;
    }

    flags.lpbk_test = 0;     /* external loopback test through transceiver */
    NO_ESI_LOOPBACK;
    Send_Lpbk_Frame();
    if (flags.lpbk_test == 0)
        Self_Test = FAILED_LPBK_TRANSCEIVER;

    flags.lpbk_mode = 0;  /* leave loopback mode */
}

Send_Lpbk_Frame()
{
    struct   TBD    *ptbd;
    int             i;

    for (i = 0; i < 8; i++) {   /* send lpbk frame 8 times, since it's
                                   best effort delivery */

#ifdef DEBUG
        if ((ptbd = Get_Tbd()) == pNULL)
            Fatal("dld - Send_Lpbk_Frame - couldn't get a TBD\n");
#else
        ptbd = Get_Tbd();

#endif /* DEBUG */

        ptbd->act_cnt = EOFBIT | LPBK_FRAME_SIZE;
        bcopy((char *) ptbd->buff_ptr, &lpbk_frame[0], LPBK_FRAME_SIZE);

        while(!Send_Frame(ptbd, &whoami[0]));
    }
}


Diagnose()
{
    struct    CB    *pcb;

#ifdef DEBUG
        if ((pcb = Get_Cb()) == pNULL)
        Fatal("dld - Diagnose - couldn't get a CB\n");
#else
        pcb = Get_Cb();

#endif /* DEBUG */

        flags.diag_done = 0;
        Self_Test = FALSE;
        pcb->cmd = DIAGNOSE | ELBIT;

        Issue_CU_Cmd(pcb);

        while (flags.diag_done == 0) ;  /* wait for Diag cmd to finish */
```

231421-35

```
/PCO/USR/CHUCK/CBRC/DLD.C

}

Configure(loopflag)

    u_short loopflag;
{
    struct  CB   *pcb;

#ifdef DEBUG
    if ((pcb = Get_Cb()) == pNULL)
        Fatal("dld - Configure - couldn't get a CB\n");
#else
    pcb = Get_Cb();

#endif /* DEBUG */


    /* Ethernet default parameters */

    pcb->parm1 = 0x080C;
    pcb->parm2 = 0x2600 | loopflag;
    pcb->parm3 = 0x6000;
    pcb->parm4 = 0xF200;
    pcb->parm5 = 0x0000;
    if (loopflag == NO_LOOPBACK)
        pcb->parm6 = 0x0040;
    else
        pcb->parm6 = 0x0006;    /* loopback frame is less bytes than
                                   the minimum frame length */
    pcb->cmd   = CONFIGURE | ELBIT;


    Issue_CU_Cmd(pcb);
}

/* Send a frame to the cable, pass a pointer to the destination address
   and a pointer to the first transmit buffer descriptor. */

Send_Frame(ptbd, padd)  /* returns false if it can't get a Command block */
struct      TBD     *ptbd;
char                *padd;
{
    struct      CB  *pcb;

    u_short         length;

    flags.reset_sema = 1;

    if ((pcb = Get_Cb()) == pNULL) {
        flags.reset_sema = 0;
        if (flags.reset_pend == 1)
            Reset_586();
        return(FALSE);
    }

    pcb->parm1 = Offset(ptbd);
```

231421-36

/PCO/USR/CHUCK/CSRC/DLD.C

```
    /* move destination address to command block */
    bcopy((char *)&pcb->parm2, (char *)padd, ADD_LEN);

    /* calculate the length field by summing up all the buffers */

    for (length = 0; ptbd->link != NULL; ptbd = Build_Ptr(ptbd->link))
        length += ptbd->act_cnt;

    length += (ptbd->act_cnt & 0x3FFF);   /* add the last buffer */

    /* check to see if padding is required, do not do padding on loopback */

    /* this will not work if MIN_DATA_LEN > TBUF_SIZE */

    if ((length < MIN_DATA_LEN) &&  /* assumes a 4 byte CRC */
        (bcmp(&whoami[0], (char *)padd, ADD_LEN) != 0))

            ptbd->act_cnt = MIN_DATA_LEN | EOFBIT;

    pcb->parm5 = length;            /* length field */


    pcb->cmd = TRANSMIT | ELBIT;


    Issue_CU_Cmd(pcb);
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(TRUE);
}


Add_Multicast_Address(pma)  /* pma - pointer to multicast address */
char    *pma;               /* returning false means the Multicast address
                               table is full */
{
    struct      MAT     *pmat;

    flags.reset_sema = 1;

/* if the multicast address is a duplicate of one already in the MAT,
   then return */

    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE &&
            (bcmp( &pmat->addr[0], (char *) pma, ADD_LEN) == 0)) {
            return(TRUE);
        }


    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if (pmat->stat == FREE) {
            pmat->stat = INUSE;
            bcopy( &pmat->addr[0], (char *) pma, ADD_LEN);
            break;
```

231421-37

/PCD/USR/CHUCK/CBRC/DLD.C

```
        }

    if (pmat > &mat[MULTI_ADDR_CNT - 1]) {
        flags.reset_sema = 0;
        if (flags.reset_pend == 1)
            Reset_586();
        return(FALSE);
    }

    Set_Multicast_Address();
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(TRUE);
}

Delete_Multicast_Address(pma)  /* returning false means the multicast address
                                  was not found */
char    *pma;
{
    struct      MAT      *pmat;

    flags.reset_sema = 1;

    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE &&
            (bcmp( &pmat->addr[0], (char *) pma, ADD_LEN) == 0)) {
            pmat->stat = FREE;
            break;
        }

    if (pmat > &mat[MULTI_ADDR_CNT - 1]) {
        flags.reset_sema = 0;
        if (flags.reset_pend == 1)
            Reset_586();
        return(FALSE);
    }

    Set_Multicast_Address();
    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
    return(TRUE);
}

Set_Multicast_Address()
{
    struct      MAT      *pmat;
    struct      MA_CB    *pma_cb;
    u_short     i;

    i = 0;
    pma_cb = &ma_cb;
    while (pma_cb->cmd != 0) ; /* if the MA_CB is inuse, wait until it's free */
    pma_cb->link = NULL;
```

231421-38

/PCD/USR/CHUCK/CBRC/DLD.C

```
    for (pmat = mat; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE) {
            bcopy( &pma_cb->mc_addr[i], &pmat->addr[0], ADD_LEN);
            i += ADD_LEN;
        }

    pma_cb->mc_cnt = i;
    pma_cb->cmd = MC_SETUP | ELBIT;

    Issue_CU_Cmd(pma_cb);

}


Put_Free_RFA(pfd)  /* Return Frame Descriptor and Receive Buffer
                      Descriptors to the Free Receive Frame Area */

    struct      FD      *pfd;
{
    struct      RBD     *prbd,  /* points to beginning of returned RBD list */
                        *q;     /* points to end of returned RBD list */
    char        ru_start_flag_fd,  /* indicates whether to restart RU */
                ru_start_flag_rbd;

    flags.reset_sema = 1;
    ru_start_flag_fd = ru_start_flag_rbd = FALSE;
    pfd->el_s = ELBIT;
    pfd->stat = 0;
    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);/* pick up the link to the rbd */
    pfd->link = pfd->rbd_offset = NULL;

    /*  Disable_586_Int(); this command is only necessary in a multitasking
    program. However in this single task environment this routine is originally
    called from isr_586(), therefore interrupts are already disabled */


    if (begin_fd == pNULL)
        begin_fd = end_fd = pfd;
    else {
        end_fd->link = Offset(pfd);
        end_fd->el_s = 0;
        end_fd = pfd;
        ru_start_flag_fd = TRUE;
    }

    if (prbd != pNULL) {        /* if there is a rbd attached to the fd then
                                find the beginning and end of the rbd list */

        for (q = prbd; q->link != NULL; q = Build_Ptr(q->link))
            q->act_cnt = 0;

            /* now prbd points to the beginning of the rbd list and
               q points to the end of the list */

        q->size = RBUF_SIZE | ELBIT;
        q->act_cnt = 0;
```

231421-39

/PCO/USR/CHUCK/CSRC/DLD.C

```
        if (begin_rbd == pNULL) { /* if there is nothing on the list
                                     create a new list */

            begin_rbd = prbd;
            end_rbd = q;
            if (prbd != q)
                ru_start_flag_rbd = TRUE; /* if there is more than one rbd
                                             returned start the RU */
        }
        else {
                                  /* if the rbd list already exists add on
                                     the new returned rbds */
            end_rbd->link = Offset(prbd);
            end_rbd->size = RBUF_SIZE;
            end_rbd = q;
            ru_start_flag_rbd = TRUE;
        }
    }
    if (ru_start_flag_fd && ru_start_flag_rbd)
        Ru_Start();

    /*  Enable_586_Int(); if Disable_586_Int() is used above */

    flags.reset_sema = 0;
    if (flags.reset_pend == 1)
        Reset_586();
}

Ru_Start()
{
    if ((scb.stat & RU_MASK) == RU_READY) /* if the RU is already 'ready'
                                             then return */
        return;

    if ((begin_fd->stat & CBIT) == CBIT)
        return;

    begin_fd->rbd_offset = Offset(begin_rbd); /* link the beginning of the rbd
                                                 list to the first fd */
    scb.rfa_offset = Offset(begin_fd);
    Wait_Scb();
    scb.cmd = RU_START;
    CA;
}

Software_Reset()
{
    scb.cmd = RESET;
    CA;
    Wait_Scb();
}

Issue_Reset_Cmds()
{
    Wait_Scb();
    scb.cmd = CU_START;
    CA;
```

231421-40

1

```
/PCD/USR/CHUCK/CSRC/DLD.C

    Wait_Scb();

    outw(0xFF5E, 0) ;       /* shut off timer 1 interrupt */
    outw(TIMER1_CNT, 0);
    outw(0xFF5E, 0xC009);   /* use timer 1 without interrupt as a deadman */

    while ((inw(0xFF5E) & 0x0020) == 0)  /* if Max Cnt bit is set before CNA
                                             is set, 586 Cmd deadlocked */
        if ((scb.stat & CNA) == CNA)
            break;

    if (scb.stat & CNA != CNA)
        Fatal("DLD:Issue_Reset_Cmds - Command deadlock during reset procedure\n");

    Reset_Timeout();

    scb.cmd = CNA;  /* Acknowledge CNA interrupt */
    CA;
    Wait_Scb();
}


/* Execute a reset, Configure, SetAddress, and MC_Setup, then restart the
    Receive Unit and the Command Unit */
Reset_586()
{
    struct      MAT     *pmat;
    u_short     i;

    ++reset_cnt;
    Disable_586_Int();
    ESI_LOOPBACK;
    Software_Reset();

    scb.stat = 0;

    CA;     /* wait for the 586 to complete initialization */


    for ( i = 0; i <= 0xFF00; i++)
        if (scb.stat == (CX | CNA))
            break;


    if (i >0xFF00)
        Fatal("DLD:init - Did not get an interrupt after Software Reset\n");

    /* Ack the reset Interrupt */
    Wait_Scb();
    scb.cmd = (CX | CNA);
    CA;
    Wait_Scb();

#ifdef  DEBUG
    if ( begin_cbl == pNULL)
        Fatal("DLD: begin_cbl = NULL in Reset_586");
#endif  /* DEBUG */
```

231421-41

```
/PCD/USR/CHUCK/CSRC/DLD.C



    /* Configure the 586 */
    /* Ethernet default parameters; Configure is not necessary when using
       default parameters  */

    res_cb.link = NULL;

    res_cb.parm1 = 0x080C;
    res_cb.parm2 = 0x2600;
    res_cb.parm3 = 0x6000;
    res_cb.parm4 = 0xF200;
    res_cb.parm5 = 0x0000;
    res_cb.parm6 = 0x0040;
    res_cb.cmd   = CONFIGURE | ELBIT;

    scb.cbl_offset = Offset(&res_cb.stat);

    Issue_Reset_Cmds();

    /* Set the Individual Address */

    bcopy((char *) &res_cb.parm1, &whoami[0], ADD_LEN); /* move the prom
                                                           address to IA cmd */
    res_cb.cmd = IA | ELBIT;

    Issue_Reset_Cmds();

    /* reload the multicast addresses */

    i = res_ma_cb.stat = 0;
    res_ma_cb.link = NULL;

    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if ( pmat->stat == INUSE ) {
            bcopy( &res_ma_cb.mc_addr[i], &pmat->addr[0], ADD_LEN);
            i += ADD_LEN;
        }

    res_ma_cb.mc_cnt = i;
    res_ma_cb.cmd = MC_SETUP | ELBIT;
    scb.cbl_offset = Offset(&res_ma_cb.stat);


    Issue_Reset_Cmds();

    /* Restart the Command Unit and the Receive Unit */

    flags.reset_sema = 0;
    flags.reset_pend = 0;

    NO_ESI_LOOPBACK;

    Recv_Int_Processing();

    scb.cbl_offset = begin_cbl;
    Wait_Scb();
```

231421-42

```
/PCO/USR/CHUCK/CSRC/DLD.C

    scb.cmd = CU_START;
    Set_Timeout();          /* Set Deadman Timer */
    CA;
    Enable_586_Int();
}

/* bcopy -- byte copy routine */
bcopy(dst, src, nbytes)
char    *dst, *src;
int     nbytes;
{
    while (nbytes--) *dst++ = *src++;
}

/* bcmp -- byte compare */
bcmp(s1, s2, nbytes)
char    *s1, *s2;
int     nbytes;
{
    while (nbytes-- && *s1++ == *s2++);
    return(*--s1 - *--s2);
}
```

231421-43

```
/PCO/USR/CHUCK/CSRC/LLC.C

/****************************************************************************
*                                                                          *
*               IEEE 802.2 Logical Link Control Layer                      *
*                       (Station Component)                                *
****************************************************************************/

#include "dld.h"

extern  char    *pNULL;

extern  struct  TBD *Get_Tbd();
extern  char        *Build_Ptr();

readonly char   xid_frame[XID_LENGTH] = { 0, 0, XID, 0x81, 0x01, 0};
/* DSAP, SSAP, XID, xid class 1 response */

struct LAT lat[DSAP_CNT];

Init_Llc()
{
    struct  LAT     *plat;

    for (plat = &lat[0]; plat <= &lat[DSAP_CNT - 1]; plat++)
        plat->stat = FREE;
    return(Init_586());
}

/* Function for adding a new DSAP */

Add_Dsap_Address(dsap, pfunc) /* DSAP must be divisible by 2**(8-N), where
                                 2**N = DSAP_CNT. (i.e. N LSBs must be 0).
                                 The function will return FALSE if does not
                                 meet the above requirements, or the Lsap
                                 Address Table is full, or the address has
                                 already been used. NULL DSAP address is
                                 reserved for the Station Component */
int dsap, (*pfunc) ();
{
    struct  LAT     *plat;

    if ((dsap << (8-DSAP_SHIFT) & 0x00FF) != 0 || dsap == 0)
        return (FALSE);

    /* Check for duplicate dsaps. */
    if ( (plat = &lat[dsap >> DSAP_SHIFT])->stat == FREE) {
            plat->stat = INUSE;
            plat->p_sap_func = pfunc;
            return (TRUE);
    }
    else
        return(FALSE);
}

/* Function for deleting DSAPs */

Delete_Dsap_Address(dsap) /* If the specified connection exists, it is severed.
                   If the connection does not exist, the command is ignored. */
```

231421-44

/PCD/USR/CHUCK/CSRC/LLC.C

```
int dsap;
{
    lat[dsap >> DSAP_SHIFT].stat = FREE;
}

Recv_Frame(pfd)
    struct  FD              *pfd;
{
    struct  RBD             *prbd;
    struct  FRAME_STRUCT    *pfs;
    struct  LAT             *plat;

    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
    pfs  = (struct FRAME_STRUCT *) prbd->buff_ptr;

    if (pfd->rbd_offset != NULL) { /* There has to be a rbd attached
                                      to the fd, or else the frame is
                                      too short. */
        if (pfs->dsap == 0) {   /* if the frame is addressed to the Station
                                   Component, then a response may be required */

            if ( !(pfs->ssap & C_R_BIT) ) {/* if the frame received is a response,
                                              instead of a command, then reject it.
                                              Because this software does not implement
                                              DUPLICATE_ADDRESS_CHECK -> no response
                                              frames should be recv'd */
                Station_Component_Response(pfd);
            }
        }
        /* not addressed to Station Component, */
        /* check to see if the dsap addressed is active */
        else if ((pfs->dsap << (8-DSAP_SHIFT) & 0x00FF) == 0 &&
                 (plat = &lat[(pfs->dsap) >> DSAP_SHIFT])->stat == INUSE ) {
            (*plat->p_sap_func)(pfd);   /* call the function associated
                                           with the dsap received */
            return;
        }
    }
    Put_Free_RFA(pfd); /* return the pfd if not given to the user saps */
}


Station_Component_Response(pfd)

    struct  FD      *pfd;
{
    struct FRAME_STRUCT     *prfs, *ptfs;
    struct TBD              *ptbd, *begin_ptbd, *q;
    struct RBD              *prbd;

    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
    prfs  = (struct FRAME_STRUCT *) prbd->buff_ptr;

    switch (prfs->cmd & ~P_F_BIT)
        {
        case    XID:
```

231421-45

/PCO/USR/CHUCK/CSRC/LLC.C

```
            while ((ptbd = Get_Tbd()) == pNULL);
            ptbd->act_cnt = EOFBIT | XID_LENGTH;
            bcopy ((char *) ptbd->buff_ptr, &xid_frame[0], XID_LENGTH);
            ptfs = (struct FRAME_STRUCT *) ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT; /* return the frame
                                                  to the sender */
            ptfs->ssap = 0;
            while(!Send_Frame(ptbd, Build_Ptr(pfd->src_addr)));
            break;

        case    TEST:

            for (prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset),
                         q = begin_ptbd = pNULL; prbd != pNULL;
                              prbd = Build_Ptr(prbd->link)) {

                while ((ptbd = Get_Tbd()) == pNULL);
                if (q != pNULL)
                    q->link = Offset(ptbd);
                else
                    begin_ptbd = ptbd;
                ptbd->act_cnt = prbd->act_cnt;
                bcopy((char *) ptbd->buff_ptr, (char *) prbd->buff_ptr,
                                                ptbd->act_cnt & 0x3FFF);
                q = ptbd;
            }

            ptfs = (struct FRAME_STRUCT *) begin_ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT;  /* return the frame to
                                                   the sender */
            ptfs->ssap = 0;
            while(!Send_Frame(begin_ptbd, Build_Ptr(pfd->src_addr)));
            break;

        }
}
```

231421-46

```
/PCO/USR/CHUCK/CSRC/UAP.C

/*****************************************************************************
*                                                                           *
*                         User Application Program                          *
*                 Async to IEEE 802.2/802.3 Protocol Converter              *
*                                                                           *
*****************************************************************************/

#include "dld.h"

/* ASCII Characters */
#define ESC      0x1B
#define LF       0x0A
#define CR       0x0D
#define BS       0x08
#define BEL      0x07
#define SP       0x20
#define DEL      0x7F
#define CTL_C        0x03

/* Hardware */
#define CH_B_CTL     0x00DE
#define CH_A_CTL     0x00DC
#define CH_B_DAT     0x00DA
#define CH_A_DAT     0x00D8
#define UART_STAT_MSK   0x70

/* Interrupt cases for 8274 */
#define UART_TX_B        0
#define UART_RECV_B      0x08
#define UART_RECV_ERR_B 0x0C
#define  EXT_STAT_INT_B 0x04
#define  EXT_STAT_INT_A 0x14

char     fifo_t[256];
char     fifo_r[256];
char     wra[5], wrb[5];
unsigned     char     in_fifo_t, out_fifo_t, in_fifo_r, out_fifo_r, actual;
u_short      t_buf_stat, r_buf_stat;

char     cbuf[80];    /* Command line buffer */
char     line[81];    /* Monitor Mode display line */

unsigned     char     dsap, ssap, send_flag, local_echo;
char     Dest_Addr[ADD_LEN];
char     Multi_Addr[ADD_LEN];

int tmstat; /* terminal mode status: for leaving terminal mode */
int dhex, monitor_flag, hs_stat;         /* flags */


extern  struct TBD      *Get_Tbd();
extern  char            *Build_Ptr();

extern  struct FLAGS    flags;

extern  char    xid_frame[],
extern  char    whoami[];
```

231421-47

```
/PCD/USR/CHUCK/CSRC/UAP.C

extern  struct  MAT     mat[];
extern  struct  LAT     lat[];
extern  char    *pNULL;

extern  unsigned long   good_xmit_cnt;
extern  u_short         underrun_cnt;
extern  u_short         no_crs_cnt;
extern  unsigned long   defer_cnt;
extern  u_short         sqe_err_cnt;
extern  u_short         max_col_cnt;
extern  unsigned long   recv_frame_cnt;
extern  u_short         reset_cnt;

extern  struct  SCB     scb;

/* Macro `type' of definitions */

#define RTS_ONB outb(CH_B_CTL,0x05);outb(CH_B_CTL,wrb[5]=wrb[5]|0x02)
#define RTS_OFFB outb(CH_B_CTL,0x05);outb(CH_B_CTL,wrb[5]=wrb[5]&0xFD)
#define RTS_ONA outb(CH_A_CTL,0x05);outb(CH_A_CTL,wra[5]=wra[5]|0x02)
#define RTS_OFFA outb(CH_A_CTL,0x05);outb(CH_A_CTL,wra[5]=wra[5]&0xFD)
#define UART_TX_DI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]&0xFD)
#define UART_TX_EI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]|0x02)
#define UART_RX_DI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]&0xE7)
#define UART_RX_EI_B outb(CH_B_CTL,0x01);outb(CH_B_CTL,wrb[1]=wrb[1]|0x10)
#define RESET_TX_INT outb(CH_B_CTL,0x28)
#define EOI_8274 outb(CH_A_CTL,0x38) /* 8274 int is IR3 on 80130 */
#define EOI_80130_8274 outb(0xE0,0x60)
#define EOI_80130_TIMER outb(0xE0,0x62)


Enable_Uart_Int()
{
    int     c;

    c = inb(0xE2);  /* read the 80130 interrupt mask register */
    outb(0xE2, 0x00FE & c); /* write to the 80130 interrupt mask register */
}

Disable_Uart_Int()
{
    int     c;

    c = inb(0xE2);
    outb(0xE2, 0x0001 | c);
}

Enable_Timer_Int()
{
    int     c;

    outb(0xEA, 125);
    outb(0xEA, 0x00); /* Timer 1 interrupts every .125 sec */
    send_flag = FALSE;
    c = inb(0xE2);  /* read the 80130 interrupt mask register */
    outb(0xE2, 0x00FB & c); /* write to the 80130 interrupt mask register */
}
```

231421-48

```
/PCO/USR/CHUCK/CSRC/UAP.C


Disable_Timer_Int()
{
    int     c;

    c = inb(0xE2);
    outb(0xE2, 0x0004 | c);
}


Co(c)
    char    c;
{
    while ( (inb(CH_B_CTL) & 4) == 0 );
    outb(CH_B_DAT, c);
}

Ci()
{
    while ( (inb(CH_B_CTL) & 1) == 0 );
    return(inb(CH_B_DAT) & 0x7F);
}

Read(pmsg, cnt, pact)
    char    *pmsg;
    unsigned char   cnt, *pact;
{
    unsigned char   i;
    char    c,  buf[200];

    for (i = c = 0; (c != CR) && (c != LF) && (i < 198); ) {
        c = Ci() & 0x7F;
        if (c == BS || c == DEL) {
            if (i > 0) {
                --i;
                Co(BS); Co(SP); Co(BS);
            }
        }
        else
            if (c >= SP) {
                Co(c);
                buf[i++] = c;
            }
        else
            if ((c == CR) || (c == LF)) {
                buf[i++] = CR;
                buf[i++] = LF;
            }
        else Co(BEL);
    }
    Co(CR); Co(LF);
    if (i > cnt)
        *pact = cnt;
    else
        *pact = i;
    for (i = 0; i < *pact ; i++)
        *pmsg++ = buf[i];
```

231421-49

/PCO/USR/CHUCK/CSRC/UAP.C

```
}

Read_Char()
{
    unsigned char   i;

    Read(&cbuf[0], 80, &actual);
    i = Skip(&cbuf[0]);
    return(cbuf[i]);
}

Write(pmsg)
    char    *pmsg;
{
    while (*pmsg != '\0') {
        if (*pmsg == '\n')
            Co(CR);
        Co(*pmsg++);
    }
}

Fatal(pmsg) /* write a message to the screen then stop */
    char    *pmsg;
{
    Write("Fatal: ");
    Write(pmsg);
    for(;;);
}

Bug(pmsg)   /* write a message to the screen then continue */
    char    *pmsg;
{
    Write("Bug: ");
    Write(pmsg);
}

Ascii_To_Char(c) /* convert ASCII-Hex to Char */
    char    c;
{
    if (('0' <= c) && (c <= '9'))
        return(c - '0');
    if (('A' <= c) && (c <= 'F'))
        return(c - 0x37);
    if (('a' <= c) && (c <= 'f'))
        return(c - 0x57);
    return(0xFF);
}

Lower_Case(c)
char    c;
{
    if (('a' <= c) && (c <= 'z'))
        return(c);
    if (('A' <= c) && (c <= 'Z'))
        return(c + 0x20);
    return(0);
}
```

231421-50

1

```
/PCO/USR/CHUCK/CSRC/UAP.C


Char_To_Ascii(c, ch) /* convert char to ASCII-Hex */
    unsigned char   c, ch[];
{
    unsigned char   i;

    i = (c & 0xF0) >> 4;
    if (i < 10)
        ch[0] = i + 0x30;
    else
        ch[0] = i + 0x37;
    i = (c & 0x0F) ;
    if (i < 10)
        ch[1] = i + 0x30;
    else
        ch[1] = i + 0x37;
    ch[2] = '\0';
}

Skip(pmsg)  /* skip blanks */
    char    *pmsg;
{
    int     i;

    for (i = 0; *pmsg == ' '; i++, pmsg++);
    return(i);
}

Read_Int()  /* Read a 16 bit Integer */
{
    u_short     wd, wh, wd1, wh1, j;
    char        i, done, hex, dover, hover;

    for (done = FALSE; done == FALSE; ) {
        Read(&cbuf[0], 80, &actual);
        i = Skip(&cbuf[0]);

        for (hex = dover = hover = FALSE, wd = wh = wd1 = wh1 = 0;
                            (j = Ascii_To_Char(cbuf[i])) <= 15; i++) {
            if (j > 9)
                hex = TRUE;
            wd = wd*10 + j;
            wh = wh*16 + j;
            if (wd < wd1)
                dover = TRUE;
            if (wh < wh1)
                hover = TRUE;
            wd1 = wd; wh1 = wh;
        }
        if (cbuf[i] == 'H' || cbuf[i] == 'h' || cbuf[i] == CR ||
                                cbuf[i] == LF || cbuf[i] == ' ') {
            if (cbuf[i] == 'H' || cbuf[i] == 'h')
                hex = TRUE;
            if (hex == TRUE && hover == FALSE)
                done = TRUE;
            if (hex == FALSE && dover == FALSE)
                done = TRUE;
```

231421-51

/PCO/USR/CHUCK/CSRC/UAP.C

```
            if (!done) {
                Write("\n This number is too big.\n It has to be less than 65536.\n"),
                Write("\n Enter number --> ");
            }
        }
        else
            Write(" Illegal Character\n Enter a number -->");
    }
    if (hex)
        return(wh);
    return(wd);
}

Int_To_Ascii(value, base, ld, ch, width) /* convert an integer to an ASCII string */
    unsigned long   value;
    u_short         base, width;
    char    ch[], ld;
{
    u_short i, j;

    for (i = 0; i < width; i++) {
        j = value % base;
        if (j < 10)   ch[i] = j + 0x30;
        else ch[i] = j + 0x37;
        value = value / base;
    }
    for (i = width - 1; ch[i] == '0' && i > 0; i--)
        ch[i] = ld;
    ch[width] = '\0';
}

Write_Long_Int(dw, i)
    unsigned long   dw;
    u_short     i;
{
    u_short     j;
    char     ch[11];

    if (dhex)
        Int_To_Ascii(dw, 16, ' ', &ch[0], 8);
    else
        Int_To_Ascii(dw, 10, ' ', &ch[0], 10);
    for (j = 0; ch[j] != '\0'; i--, j++)
        line[i] = ch[j];
}

Write_Short_Int(w, i)
    u_short w, i;
{
    u_short j;
    char    ch[6];
    unsigned long   dw;

    dw = w;
    if (dhex)
        Int_To_Ascii(dw, 16, '0', &ch[0], 4);
    else
```

231421-52

1

/PCO/USR/CHUCK/CSRC/UAP.C

```
        Int_To_Ascii(dw, 10, '0', &ch[0], 5);
    for (j = 0; ch[j] != '\0'; i--, j++)
        line[i] = ch[j];
}


Yes()
{
    char    b;

    for ( ; ; ) {
        b = Read_Char();
        if ((b == 'Y') || (b == 'y'))
            return(TRUE);
        if ((b == 'N') || (b == 'n'))
            return(FALSE);
        Write(" Enter a Y or N --> ");
    }
}

Read_Addr(pmsg, add, cnt) /* pmsg - pointer to the output message */
                          /* add - pointer to the address */
                          /* cnt - number of bytes in the address */
    char    *pmsg, add[], cnt;
{
    char    i, j;

    for ( ; ; ) {
        Write(pmsg);
        Read(&cbuf[0], 80, &actual);
        for (j = skip(&cbuf[0]), i = 0; i < 2*cnt ; i++, j++) {
            if (('0' <= cbuf[j]) && (cbuf[j] <= '9'))
                cbuf[i] = cbuf[j] - '0';
            else
                if (('A' <= cbuf[j]) && (cbuf[j] <= 'F'))
                    cbuf[i] = cbuf[j] - 0x37;
            else
                if (('a' <= cbuf[j]) && (cbuf[j] <= 'f'))
                    cbuf[i] = cbuf[j] - 0x57;
            else {
                Write(" Illegal Character\n");
                break;
            }
        }
        if (i >= 2*cnt - 1)
            break;
    }
    for (i = 0; i <= cnt - 1; i++)
        add[(cnt - 1) - i] = cbuf[2*i] << 4 | cbuf[2*i + 1];
}

Write_Addr(padd, cnt)
    char    padd[], cnt;
{
    unsigned char   i, c[3];

    for ( ; cnt >0 ; cnt--) {
```

231421-53

```
/PCD/USR/CHUCK/CBRC/UAP.C

        i = padd[cnt-1];
        Char_To_Ascii(i, &c[0]);
        Write(&c[0]);
    }
    c[0] = '\n';
    c[1] = '\0';
    Write(&c[0]);
}

Recv_Data_1(pfd)     /* Receives the frame from the 802.2 module */

    struct  FD                *pfd;
{
    struct FRAME_STRUCT       *prfs, *ptfs;
    struct TBD                *ptbd, *begin_ptbd, *q;
    struct RBD                *prbd;
    char                      *prbuf;
    int                       cnt;

    prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset);
    prfs  = (struct FRAME_STRUCT *) Build_Ptr(prbd->buff_ptr);

    switch (prfs->cmd & ~P_F_BIT)   {
        case    UI:
            if (monitor_flag)
                break;  /* Don't put data in fifo unless in terminal mode */
            prbuf = (char *) prfs;
            prbuf += 3; /* skip over the header info and point to the data */
            cnt = 3;
            pfd->length -= 3;
            for (; prbd != pNULL; cnt = 0, prbuf = (char *) prbd->buff_ptr){
                for ( ; cnt < (prbd->act_cnt & 0x03FFF) && pfd->length > 0;
                                        cnt++, prbuf++, pfd->length--) {
                    while(r_buf_stat == FULL);
                    Fifo_R_In(*prbuf);
                }
                prbd = Build_Ptr(prbd->link);
#ifdef DEBUG
                if (pfd->length == 0 && prbd != pNULL)
                    Fatal("Uap: Recv_Data_1(pfd) ");
#endif /* DEBUG */
            }
            break;

        case    XID:

            while ((ptbd = Get_Tbd()) == pNULL);
            ptbd->act_cnt = EOFBIT | XID_LENGTH;
            bcopy ((char *) ptbd->buff_ptr, &xid_frame[0], XID_LENGTH);
            ptfs = (struct FRAME_STRUCT *) ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT; /* return the frame
                                                  to the sender */
            ptfs->ssap = ssap;
            while(!Send_Frame(ptbd, Build_Ptr(pfd->src_addr)));
```

231421-54

1

/PCO/USR/CHUCK/CSRC/UAP.C

```
            break;

        case    TEST:

            for (prbd = (struct RBD *) Build_Ptr(pfd->rbd_offset),
                            q = begin_ptbd = pNULL; prbd != pNULL;
                                prbd = Build_Ptr(prbd->link)) {
                while ((ptbd = Get_Tbd()) == pNULL);
                if (q != pNULL)
                    q->link = Offset(ptbd);
                else
                    begin_ptbd = ptbd;
                ptbd->act_cnt = prbd->act_cnt;
                bcopy((char *) ptbd->buff_ptr, (char *) prbd->buff_ptr,
                                        ptbd->act_cnt & 0x3FFF);
                q = ptbd;
            }

            ptfs = (struct FRAME_STRUCT *) begin_ptbd->buff_ptr;
            ptfs->cmd = prfs->cmd;

            ptfs->dsap = prfs->ssap | C_R_BIT;  /* return the frame to
                                                   the sender */
            ptfs->ssap = ssap;
            while(!Send_Frame(begin_ptbd, Build_Ptr(pfd->src_addr)));
            break;

    }
    Put_Free_RFA(pfd);  /* return the frame */
}


Fifo_T_Out()    /* called by main program */
{
    char    c;

    c = fifo_t[out_fifo_t++];

    Disable_Uart_Int();
    if (out_fifo_t == in_fifo_t)    /* if the fifo is empty */
        t_buf_stat = EMPTY; /* stop filling Transmit Buffer Descriptors */
    else        /* if the fifo was full and is now draining */
        if (t_buf_stat == FULL && out_fifo_t - 80 == in_fifo_t) { /* turn on
                                                        the spigot */
            RTS_ONB;
            t_buf_stat = INUSE;
        }
    Enable_Uart_Int();
    return(c);
}

Fifo_T_In(c)    /* called by Uart receive interrupt */
    char    c;
{

    fifo_t[in_fifo_t++] = c;
    if (t_buf_stat == EMPTY)
```

231421-55

```
/PCD/USR/CHUCK/CSRC/UAP.C

        t_buf_stat = INUSE; /* start filling Transmit Buffer Descriptor */
    else        /* if there are only 20 locations left, turn off the spigot */
        if (t_buf_stat == INUSE && in_fifo_t + 20 == out_fifo_t) {
            RTS_OFFB;
            t_buf_stat = FULL;
        }
}

Fifo_R_Out()    /* called by transmit interrupt */
{
    char    c;

    c = fifo_r[out_fifo_r++];

    if (out_fifo_r == in_fifo_r)    /* if the fifo is empty */
        r_buf_stat = EMPTY;
    else        /* if the fifo was full and is now draining */
        if (r_buf_stat == FULL && out_fifo_r - 81 == in_fifo_r)
            r_buf_stat = INUSE;
    return(c);
}

Fifo_R_In(c)    /* called by Recv_Data_1() */
    char    c;
{
    fifo_r[in_fifo_r++] = c;
    Disable_Uart_Int();
    if (r_buf_stat == EMPTY) {
        UART_TX_EI_B;
        Co(0);              /* prime the interrupt */
        r_buf_stat = INUSE;
    }
    else        /* if the buffer is full, indicate it */
        if (r_buf_stat == INUSE && in_fifo_r == out_fifo_r)
            r_buf_stat = FULL;
    Enable_Uart_Int();
}

Isr_Uart()
{
    int     stat;
    char    c;

    outb(CH_B_CTL, 2);  /* point to RR2 in 8274 */

    switch(inb(CH_B_CTL) & 0x1C){ /* read 8274 interrupt vector and service it */

    case UART_TX_B:

        if (r_buf_stat == EMPTY) {
            UART_TX_DI_B;           /* if fifo is empty disable transmitter */
            RESET_TX_INT;
        }
        else
            outb (CH_B_DAT, Fifo_R_Out());
        break;
```

231421-56

```
/PCO/USR/CHUCK/CSRC/UAP.C

    case UART_RECV_ERR_B:

        outb(CH_B_CTL, 1); /* point to RR1 in 8274 */
        stat = inb(CH_B_CTL);
        outb(CH_B_CTL, 0x30);
        if (stat & 0x0010)
            Write("\nParity Error Detected\n");
        if (stat & 0x0020)
            Write("\nOverrun Error Detected\n");
        if (stat & 0x0040)
            Write("\nFraming Error Detected\n");
        break;

    case UART_RECV_B:

        c = inb(CH_B_DAT);

        if (hs_stat == TRUE) {
            hs_stat = FALSE;  /* Flag to terminate High Speed Transmit mode */
            break;
        }

        if (local_echo)
            Co(c);            /* echo the char back to the terminal; could cause
                                 a transmit overrun if Tx interrupt is enabled */
        if (c == CTL_C)
            tmstat = FALSE;
        else
            Fifo_T_In(c);
        break;

    case EXT_STAT_INT_B:

        outb(CH_B_CTL, 0x10);   /* reset external status interrupts */
        break;

    case EXT_STAT_INT_A:

        outb(CH_A_CTL, 0x10);
        break;


    default:
        ;
    }
    EOI_80130_8274;
    EOI_8274;
}

Isr2()
{
    send_flag = TRUE;
    outb(0xEA, 125);
    outb(0xEA, 0x00); /* Timer 1 interrupts every .125 sec */
    outb(0xE0, 0x62);   /* EOI 80130 */
}
```

231421-57

```
/PCO/USR/CHUCK/CBRC/UAP.C

Load_Lsap()
{
    int     Recv_Data_1();

    for(;;) {
        Read_Addr("\n\nEnter this Station's LSAP in Hex --> ", &ssap, 1);
        if (!Add_Dsap_Address(ssap, Recv_Data_1)) {
            Write("\n\nError: LSAP Address must be one of the following:\n");
            Write("\n       20H, 40H, 60H, 80H, A0H, C0H, E0H \n");
        }
        else break;
    }
}

Load_Multicast()
{

    for ( ; ; ) {
        Read_Addr("\nEnter the Multicast Address in Hex -->",
                                                  &Multi_Addr[0], ADD_LEN);
        if ((Multi_Addr[0] & 0x01) == 0)
            Write("\nSorry, the LSB of the Multicast Address must be 1\n");
        else { if (!Add_Multicast_Address(&Multi_Addr[0])) {
                    Write("\n\nSorry, Multicast Address Table is full!\n");
                    break;
                }
                else {
                    Write("\n\nWould you like to add another Multicast Address?");
                    Write(" (Y or N) --> ");
                    if (!Yes())
                        break;
                }
        }
    }
}

Remove_Multicast()
{

    for ( ; ; ) {
        Read_Addr("\nEnter the Multicast Address that you want to delete in Hex -->",
                                                  &Multi_Addr[0], ADD_LEN);
        if ((Multi_Addr[0] & 0x01) == 0)
            Write("\nSorry, the LSB of the Multicast Address must be 1\n");
        else { if (!Delete_Multicast_Address(&Multi_Addr[0])) {
                    Write("\n\nSorry, that Multicast Address doesn't exist!\n");
                    break;
                }
                else {
                    Write("\n\nWould you like to delete another Multicast Address?");
                    Write(" (Y or N) --> ");
                    if (!Yes())
                        break;
                }
        }
    }
}
```

231421-58

```
/PCD/USR/CHUCK/CSRC/UAP.C


Print_Addresses()
{
    struct  MAT *pmat;
    int         stat;

    Write("\n This Stations Host Address is: ");
    Write_Addr(&whoami[0], ADD_LEN);
    Write("\n The Address of the Destination Node is: ");
    Write_Addr(&Dest_Addr[0], ADD_LEN);
    Write("\n This Stations LSAP Address is: ");
    Write_Addr(&ssap, 1);
    Write("\n The Address of the Destination LSAP is: ");
    Write_Addr(&dsap, 1);
    stat = FALSE;
    for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
        if (pmat->stat == INUSE) {
            stat = TRUE;
            break;
        }
    if (stat)   {
        Write("\n The following Multicast Addresses are enabled: ");
        for (pmat = &mat[0]; pmat <= &mat[MULTI_ADDR_CNT - 1]; pmat++)
            if (pmat->stat == INUSE) {
                Write_Addr(&pmat->addr[0], ADD_LEN);
                Write("                                            ");
            }
    }
    else
        Write("\n There are no Multicast Addresses enabled.\n");
}

Init_DataLink()
{
    int     stat;

    if ((stat = Init_Llc()) == PASSED)
        Write("\n\nPassed Diagnostic Self Tests\n\n\n");
    else
        if(stat == FAILED_DIAGNOSE)
            Write("\n\nFailed: Self Test Diagnose Command\n");
    else
        if(stat == FAILED_LPBK_INTERNAL)
            Write("\n\nFailed: Internal Loopback Self Test\n");
    else
        if(stat == FAILED_LPBK_EXTERNAL)
            Write("\n\nFailed: External Loopback Self Test\n");
    else
        if(stat == FAILED_LPBK_TRANSCEIVER)
            Write("\n\nFailed: External Loopback Through Transceiver Self Test\n");
}

Init_Uap()
{

    outb(0xE0, 0x31);    /*initalize 80130 pic - ICW1 */
    outb(0xE2, 0x20);    /* ICW2 */
```

231421-59

/PCO/USR/CHUCK/CSRC/UAP.C

```
    outb(0xE2, 0x10);    /* ICW3 */
    outb(0xE2, 0x0D);    /* ICW4 */
    outb(0xE2, 0x10);    /* ICW6 */
    outb(0xE2, 0xFF);    /* mask all interrupts */

    outw(0xFF20, 0x0020);    /* set 80186 vector base */

    /* Initialize the 80130 timers for Terminal Mode */

    outb(0xEE, 0x34);
    outb(0xE8, 0x88);
    outb(0xE8, 0x08); /* SYSTICK set for 1 msec */
    outb(0xEE, 0x70);
    outb(0xEA, 125);
    outb(0xEA, 0x00); /* Timer 1 interrupts every .125 sec */

    /* Initialize the 8274 */
    outb(CH_B_CTL, 0x10); outb(CH_B_CTL, 0x28); outb(CH_B_CTL, 0x30);
    outb(CH_A_CTL, 0x38);
    outb(CH_B_CTL, 2); outb(CH_B_CTL, wrb[2] = 0x14);
    outb(CH_B_CTL, 1); outb(CH_B_CTL, wrb[1] = 0x15);
    outb(CH_B_CTL, 5); outb(CH_B_CTL, wrb[5] = 0xEA);

    Write("\n\n\n\n\n\n\n\n\n\n\n\n");
    Write("             **********************************************************\n");
    Write("             * 82586 IEEE 802.2/802.3 Compatible Data Link Driver *\n");
    Write("             **********************************************************\n");
    Write("\n\n\n\n\n\n\n");

    Init_DataLink();

    dhex = FALSE;
    monitor_flag = TRUE;

    Read_Addr("\n\nEnter the Address of the Destination Node in Hex --> ",
                                            &Dest_Addr[0], ADD_LEN);
    Load_Lsap();

    Read_Addr("\n\nEnter the Destination Node's LSAP in Hex --> ", &dsap, 1);

    Write("\n\nDo you want to Load any Multicast Addresses? (Y or N) -->");

    if (Yes())
        Load_Multicast();

    Print_Addresses();
}

Terminal_Mode()
{
    int       frame_cnt, buf_cnt;
    struct    TBD       *ptbd, *q, *begin_ptbd;
    char                *pbuf, c;

    Write("\n Would you like the local echo on? (Y or N)-->");

    if(Yes())
```

231421-60

/PCO/USR/CHUCK/CSRC/UAP.C

```
        local_echo = TRUE;
    else
        local_echo = FALSE;

    Write("\n This program will now enter the terminal mode.\n\n");
    Write("\n Press ^C then CR to return back to the menu\n\n");

    /* Initialize Fifo variables */

    out_fifo_t = in_fifo_t = out_fifo_r = in_fifo_r = 0;
    t_buf_stat = EMPTY;  r_buf_stat = EMPTY;

    EOI_80130_8274;
    Enable_Uart_Int();
    Enable_Timer_Int();
    monitor_flag = FALSE;
    tmstat = TRUE;
    while (tmstat)  {

        for (frame_cnt = 0; frame_cnt < MAX_FRAME_SIZE;   q = ptbd) {

            while ((ptbd = Get_Tbd()) == pNULL); /* get a xmit buffer from the
                                                    data link */
            pbuf = (char *) ptbd->buff_ptr;  /* point to the buffer */
            buf_cnt = 0;

            if (frame_cnt == 0) { /* if this is the first buffer, add on IEEE 802.2
                                    header information */
                begin_ptbd = ptbd;
                *pbuf++ = dsap;
                *pbuf++ = ssap;
                *pbuf++ = UI;
                buf_cnt = 3;
            }
            else  q->link = Offset(ptbd); /* if this isn't the first buffer
                            link the previous buffer with the new one */
            /* fill up a data link xmit buffer from async transmit fifo */
            for ( ; buf_cnt < TBUF_SIZE && frame_cnt < MAX_FRAME_SIZE;
                                    buf_cnt++, pbuf++, frame_cnt++) {
                if (frame_cnt != 0 && send_flag)
                    break;

                while (t_buf_stat == EMPTY);     /* wait until fifo has data */
                if ((c = *pbuf = Fifo_T_Out()) == CR) {
                    ++buf_cnt ; ++pbuf; ++frame_cnt;
                    break;
                }
            }
            if (c == CR || buf_cnt < TBUF_SIZE || send_flag) { /* last buffer in list */
                ptbd->act_cnt = buf_cnt | EOFBIT;
                send_flag = FALSE;
                break;
            }
        }
        while(!Send_Frame(begin_ptbd, &Dest_Addr[0])); /* keep trying until
                                                         successful */
    }
```

231421-61

/PCD/USR/CHUCK/CSRC/UAP.C

```
    Disable_Uart_Int();
    Disable_Timer_Int();
    monitor_flag = TRUE;
}


struct  TBD     *Build_Frame(cnt)
    u_short     cnt;
{
    u_short         buf_cnt, frame_cnt, i;
    struct  TBD     *ptbd, *q, *begin_ptbd;
    char            *pbuf;

    i = 0x20; frame_cnt = 0;

    for ( ; ; q = ptbd) {

        while ((ptbd = Get_Tbd()) == pNULL); /* get a xmit buffer from the
                                                data link */

        pbuf = (char *) ptbd->buff_ptr;  /* point to the buffer */
        buf_cnt = 0;

        if (frame_cnt == 0) { /* if this is the first buffer, add on IEEE 802.2
                                header information */
            begin_ptbd = ptbd;
            *pbuf++ = dsap;
            *pbuf++ = ssap;
            *pbuf++ = UI;
            buf_cnt = 3;
        }
        else  q->link = Offset(ptbd); /* if this isn't the first buffer
                            link the previous buffer with the new one */
        /* fill up a data link xmit buffer with ASCII characters */
        for (; buf_cnt < TBUF_SIZE && cnt > 0;
                            i++, buf_cnt++, pbuf++, cnt--, frame_cnt++) {
            *pbuf = i;
            if (i > 0x7E)
                i = 0x1F;
        }
        if (cnt == 0) { /* last buffer in list */
            ptbd->act_cnt = buf_cnt | EOFBIT;
            break;
        }
    }
    return(begin_ptbd);
}


Monitor_Mode()
{
    u_short     xmit, cnt, i;
    struct  TBD     *Build_Frame(), *ptbd;

    Write(" Do you want this station to transmit? (Y or N) --> ");
    if (Yes())
```

231421-62

```
/PCO/USR/CHUCK/CSRC/UAP.C

        for (xmit = FALSE; xmit == FALSE; ) {
        Write("\n Enter the number of data bytes in the frame --> ");
        cnt = Read_Int();
        if (cnt > 2045)
            Write ("\n Sorry, the number has to be less than 2046!\n");
        else
            xmit = TRUE;
        }

    else xmit = FALSE;

    Write("\n Hit any key to exit Monitor Mode.\n\n");

Write("  # of Good      # of Good     CRC      Alignment     No           Receive\n");
Write("   Frames         Frames      Errors     Errors       Resource     Overrun\n");
Write(" Transmitted     Received                             Errors       Errors\n");

   /* "01234567890123456789012345678901234567890123456789012345678901234567890123456789012345678
         xxxxxxxxxx    xxxxxxxxxx    xxxx        xxxx          xxxx           xxxx
           xxxxxxxx      xxxxxxxx
                 11              25      33          44            57              71 */

    for (i = 0; i < 79; i++)
        line[i] = 0x20;
    line[79] = CR;
    line[80] = '\0';

    while ((inb(CH_B_CTL) & 1) == 0) {
        for (i = 0; i < 72; i++)
            line[i] = 0x20;
        Write_Long_Int(good_xmit_cnt, 11);
        Write_Long_Int(recv_frame_cnt, 25);
        Write_Short_Int(scb.crc_errs, 33);
        Write_Short_Int(scb.aln_errs, 44);
        Write_Short_Int(scb.rsc_errs, 57);
        Write_Short_Int(scb.ovr_errs, 71);
        Write(&line[0]);
        if (xmit) {
            ptbd = Build_Frame(cnt);
            while(!Send_Frame(ptbd, &Dest_Addr[0]));
        }
    }
    i = Ci();
}

Hs_Xmit_Mode()
{
    struct  TBD  *ptbd;

    Write("\n Hit any key to exit High Speed Transmit Mode.\n\n");

    hs_stat = TRUE;
    EOI_80130_8274;
    Enable_Uart_Int();

    /* Execute this loop until a recv char interrupt happends at Uart */
```

231421-63

```
/PCO/USR/CHUCK/CSRC/UAP.C

    while (hs_stat) {
        while ((ptbd = Get_Tbd()) == pNULL); /* get a xmit buffer from
                                                the data link */
        ptbd->act_cnt |= EOFBIT;           /* set the End Of Frame bit */
        while(!Send_Frame(ptbd, &Dest_Addr[0])); /* Send Frame */
    }

    Disable_Uart_Int();
}

Print_Cnt()
{
    char    ch[11], base, dwidth, width, i;
    unsigned    long    temp;

      (dhex) {
        dwidth = 8;
        width = 4;
        base = 16;
    }
    else {
        base = 10;
        dwidth = 10;
        width = 5;
    }

    Write("\n\n Good frames transmitted: ");
    for (i = 1; i <= 11 - dwidth; i++)
        Co(SP);
    Int_To_Ascii(good_xmit_cnt, base, ' ', &ch[0], dwidth);
    for (i = dwidth - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Good frames received: ");
    for (i = 1; i <= 15 - dwidth; i++)
        Co(SP);
    Int_To_Ascii(recv_frame_cnt, base, ' ', &ch[0], dwidth);
    for (i = dwidth - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n CRC errors received: ");
    for (i = 1; i <= 15 - width; i++)
        Co(SP);
    temp = scb.crc_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Alignment errors received: ");
    for (i = 1; i <= 10 - width; i++)
        Co(SP);
    temp = scb.aln_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n Out of Resource frames: ");
    for (i = 1; i <= 12 - width; i++)
        Co(SP);
    temp = scb.rsc_errs;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
```

231421-64

```
/PCD/USR/CHUCK/CBRC/UAP.C

    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Receiver overrun frames: ");
    for (i = 1; i <= 12 - width; i++)
        Co(SP);
    temp = scb.ovr_errs;
    int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n 82586 Reset: ");
    for (i = 1; i <= 23 - width; i++)
        Co(SP);
    temp = reset_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Transmit underrun frames: ");
    for (i = 1; i <= 11 - width; i++)
        Co(SP);
    temp = underrun_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n Lost CRS: ");
    for (i = 1; i <= 26 - width; i++)
        Co(SP);
    temp = no_crs_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  SQE errors: ");
    for (i = 1; i <= 25 - width; i++)
        Co(SP);
    temp = sqe_err_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("\n\n Maximum retry: ");
    for (i = 1; i <= 21 - width; i++)
        Co(SP);
    temp = max_col_cnt;
    Int_To_Ascii(temp, base, ' ', &ch[0], width);
    for (i = width - 1; i >= 0; i--)
        Co(ch[i]);
    Write("  Frames that deferred: ");
    for (i = 1; i <= 15 - dwidth; i++)
        Co(SP);
    Int_To_Ascii(defer_cnt, base, ' ', &ch[0], dwidth);
    for (i = dwidth - 1; i >= 0; i--)
        Co(ch[i]);
}


Print_Help()
{
    Write ("\n\n Commands are:\n\n");
    Write (" T - Terminal Mode                    M - Monitor Mode\n");
```

231421-65

```
/PCD/USR/CHUCK/CSRC/UAP.C

    Write (" X - High Speed Transmit Mode         V - Change Transmit Statistics\n");
    Write (" P - Print All Counters               C - Clear All Counters\n");
    Write (" A - Add a Multicast Address          Z - Delete a Multicast Address\n");
    Write (" S - Change the SSAP Address          D - Change the DSAP Address\n");
    Write (" N - Change Destination Node Address  L - Print All Addresses\n");
    Write (" R - Re-Initialize the Data Link      B - Change the number Base\n");
}

Main()
{
    int     c;

    Init_Uap();
    Print_Help();

    for (;;) {
        Write ("\n\n Enter a command, type H for Help --> ");
        c = Read_Char();
        switch (Lower_Case(c)) {

        case 'h':
            Print_Help();
            break;
        case 'm':
            Monitor_Mode();
            break;
        case 't':
            Terminal_Mode();
            break;
        case 'x':
            Hs_Xmit_Mode();
            break;
        case 'v':
            Write("\n Transmit Statistics are now ");
            if (flags.stat_on == 1)
                Write("on.\n Would you like to change it ? (Y or N) --> ");
            else
                Write("off.\n Would you like to change it ? (Y or N) --> ");
            if (Yes()) {
                if (flags.stat_on == 1)
                    flags.stat_on = 0;
                else flags.stat_on = 1;
            }
            break;
        case 'p':
            Print_Cnt();
            break;
        case 'c':
            Clear_Cnt();
            break;
        case 'a':
            Load_Multicast();
            break;
        case 'z':
            Remove_Multicast();
            break;
        case 's'
```

231421-66

/PCO/USR/CHUCK/CSRC/UAP.C

```
            Delete_Dsap_Address(ssap);
            Load_Lsap();
            break;
        case 'd':
            Read_Addr("\n\nEnter the Destination Node's LSAP in Hex --> ", &dsap, 1);
            break;
        case 'n':
            Read_Addr("\n\nEnter the Address of the Destination Node in Hex --> ",
                                                   &Dest_Addr[0], ADD_LEN);
            break;
        case 'l':
            Print_Addresses();
            break;
        case 'r':
            Software_Reset();
            Init_DataLink();
            Add_Dsap_Address(ssap, Recv_Data_1);
            break;
        case 'b':
            Write("\n The current base is ");
            if (dhex == TRUE)
                Write("Hex.\n Would you like to change it ? (Y or N) --> ");
            else
                Write("Decimal.\n Would you like to change it ? (Y or N) --> ");
            if (Yes()) {
                if (dhex == TRUE)
                    dhex = FALSE;
                else dhex = TRUE;
            }
            break;

        default:
            Write ("\n Unknown command\n");
            break;
        }
    }
}
```

231421-67

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

name    c_assy_support

stack   segment stack   'stack'
stktop  label   word
stack   ends

DLD_DATA                segment public  'DATA'
extrn   SEGMT_:word                     ; data segment address
DLD_DATA                ends

UAP_DATA        segment public   'DATA'
UAP_DATA        ends

DLD_CODE        segment public  'CODE'
        extrn    Isr_Timeout_:far, Isr_586_:far, Isr7_:far
        extrn    Isr6_:far, Isr5_:far, Isr1_:far
DLD_CODE        ends

UAP_CODE        segment public  'CODE'
        extrn   Isr_Uart_:far, Isr2_:far, Main_:far
UAP_CODE        ends


DG_CODE         segment public  'CODE'

        public   inw_, outw_, init_intv_, enable_, disable_, Build_Ptr_
        public   Offset_, begin, inb_, outb_

arg1    equ     [BP + 6]
arg2    equ     [BP + 8]

        assume  CS:DG_CODE
        assume  DS:DLD_DATA

;+
;       initialization program for the 82586 data link driver
;-

begin:

        sti
        mov     ax, DLD_DATA ;get base of dgroup and
        mov     SEGMT_, ax      ;pass the segment value to the c program
        mov     ds, ax
        call Main_              ;go to the c program
        hlt


inb_    proc    far
        push    BP
        mov     BP, SP
        push    DX
        mov     DX, arg1
        in      AL, DX
        pop     DX
        mov     SP, BP
```

231421-68

/PCD/USR/CHUCK/CSRC/ASSY.ASM

```
        pop     BP
        ret
inb_    endp

outb_   proc    far
        push    BP
        mov     BP, SP
        push    DX
        push    AX
        mov     DX, arg1
        mov     AX, arg2
        out     DX, AL
        pop     AX
        pop     DX
        mov     SP, BP
        pop     BP
        ret
outb_   endp

inw_    proc    far
        push    BP
        mov     BP, SP
        push    DX
        mov     DX, arg1
        in      AX, DX
        pop     DX
        mov     SP, BP
        pop     BP
        ret
inw_    endp

outw_   proc    far
        push    BP
        mov     BP, SP
        push    DX
        push    AX
        mov     DX, arg1
        mov     AX, arg2
        out     DX, AX
        pop     AX
        pop     DX
        mov     SP, BP
        pop     BP
        ret
outw_   endp

Build_Ptr_      proc    far
        push    BP
        mov     BP, SP
        mov     DX, DLD_DATA
        mov     AX, arg1
        mov     SP, BP
        pop     BP
        ret
Build_Ptr_      endp

Offset_ proc    far
```

231421-69

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

        push    BP
        mov     BP, SP
        mov     AX, arg1
        mov     SP, BP
        pop     BP
        ret
Offset_ endp

serve_int_isr   proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr_586_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int_isr   endp

serve_int_8274  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, UAP_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr_Uart_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
```

231421-70

```
/PCD/USR/CHUCK/CBRC/ASBY.ASM

        pop     CX
        pop     BX
        pop     AX
        iret
serve_int_8274  endp

serve_int_timeout       proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr_Timeout_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int_timeout       endp

serve_int7_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr7_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
```

231421-71

```
/PCD/USR/CHUCK/CSRC/ASSY.ASM

        iret
serve_int7_isr  endp

serve_int6_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr6_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int6_isr  endp

serve_int5_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr5_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int5_isr  endp
```

231421-72

/PCO/USR/CHUCK/CBRC/ASSY.ASM

```
serve_int2_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, UAP_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr2_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int2_isr  endp

serve_int1_isr  proc    far
        push    AX
        push    BX
        push    CX
        push    DX
        push    SI
        push    DI
        push    DS
        push    ES

        mov     AX, DLD_DATA
        mov     DS, AX
        mov     ES, AX

        call    Isr1_

        pop     ES
        pop     DS
        pop     DI
        pop     SI
        pop     DX
        pop     CX
        pop     BX
        pop     AX
        iret
serve_int1_isr  endp


enable_ proc    far
        sti
```

231421-73

```
/PCO/USR/CHUCK/CSRC/ASSY.ASM

        ret
enable_ endp

disable_        proc    far
        cli
        ret
disable_        endp

init_intv_      proc    far
        push    DS
        push    AX

        xor     AX, AX
        mov     DS, AX

        ; Interrupt types for the 186/51 COMMputer

        mov     DS:word ptr  80h, offset serve_int_8274        ; int 0
        mov     DS:word ptr  82h, DG_CODE
        mov     DS:word ptr  84h, offset serve_int1_isr        ; int 1
        mov     DS:word ptr  86h, DG_CODE
        mov     DS:word ptr  88h, offset serve_int2_isr        ; int 2
        mov     DS:word ptr  8Ah, DG_CODE
        mov     DS:word ptr  8Ch, offset serve_int_isr         ; int 3
        mov     DS:word ptr  8Eh, DG_CODE
        mov     DS:word ptr  90h, offset serve_int_timeout     ; int 4
        mov     DS:word ptr  92h, DG_CODE
        mov     DS:word ptr  94h, offset serve_int5_isr        ; int 5
        mov     DS:word ptr  96h, DG_CODE
        mov     DS:word ptr  98h, offset serve_int6_isr        ; int 6
        mov     DS:word ptr  9Ah, DG_CODE
        mov     DS:word ptr  9Ch, offset serve_int7_isr        ; int 7
        mov     DS:word ptr  9Eh, DG_CODE


        pop     AX
        pop     DS
        ret

init_intv_      endp

DG_CODE ends
        end             begin, ds:dld_data, ss:stack:stktop
```

231421-74